Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jakob Maležič

Neprekinjena integracija in dostava poslovno kritičnih aplikacij

magistrsko delo

magistrski študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Nejc Ilc
Somentor: dr. Tadej Justin

Ljubljana, 2023

[1]V dogovorju z mentorjem lahko kandidat magistrsko delo s pripadajočo izvorno kodo izda tudi pod drugo licenco, ki ponuja določen del pravic vsem: npr. Creative Commons, GNU GPL. V tem primeru na to mesto vstavite opis licence, na primer tekst [1].

Zahvala

Na tem mestu zapišite, komu se zahvaljujete za izdelavo magistrske naloge. V zahvali se poleg mentorja spodobi omeniti vse, ki so s svojo pomočjo prispevali k nastanku vašega izdelka.

Jakob Maležič, 2023

Vsem rožicam tega sveta.

"The only reason for time is so that everything doesn't happen at once."

— Albert Einstein

Kazalo

Seznam uporabljenih kratic

kratica	**angleško**	**slovensko**
CA	classification accuracy	klasifikacijska točnost
DBMS	database management system	sistem za upravljanje podatkovnih baz
SVM	support vector machine	metoda podpornih vektorjev
...	...	...

Povzetek

Naslov: Neprekinjena integracija in dostava poslovno kritičnih aplikacij

V vzorcu je predstavljen postopek priprave magistrskega dela z uporabo okolja LaTeX. Vaš povzetek mora sicer vsebovati približno 100 besed, ta tukaj je odločno prekratek. Dober povzetek vključuje: (1) kratek opis obravnavanega problema, (2) kratek opis vašega pristopa za reševanje tega problema in (3) (najbolj uspešen) rezultat ali prispevek magistrske naloge.

Ključne besede

neprekinjena integracija, neprekinjena dostava, poslovno kritične aplikacije

Abstract

Title: Continuous integration and delivery for business critical applications

This sample document presents an approach to typesetting your BSc thesis using LaTeX. A proper abstract should contain around 100 words which makes this one way too short. A good abstract contains: (1) a short description of the tackled problem, (2) a short description of your approach to solving the problem, and (3) (the most successful) result or contribution in your thesis.

Keywords

continuous integration, continuous deployment, business critical applications

Poglavje 1

Uvod

Pri razvoju večje programske opreme ali aplikacije je programje navadno razdeljeno na več zaključenih enot, ki jih različne razvojne skupine ali razvojna podjetja neodvisno razvijajo. Za slednje je pred delitvijo nalog vnaprej izbrano in specificirano tudi produkcijsko okolje. Na ta način si lahko razvojne skupine, za preverjanje realizacije izvedenega dela, pripravijo razvojno okolje, ki ima čim bolj podobno infrastrukturo, kot je na voljo v produkcijskem okolju. Samo tako so posamezne skupine lahko dovolj samozavestne, da bo njihovo programje pravilno delovalo tudi v produkcijskem okolju.

1.1 Motivacija

Poseben primer produkcijskega okolja predstavlja zaprto produkcijsko okolje stranke, ki se velikokrat pojavi pri poslovno kritičnih aplikacijah. Poslovno kritične aplikacije so tiste aplikacije, ki so ključne za delovanje poslovnega procesa podjetja [2, 3]. V takšnem primeru je zaradi varnosti produkcijsko okolje pogosto razvijalcem nedostopno, kot tudi celotno omrežje stranke. S tem je dostava programske kode v omrežje stranke s strani razvijalcev nemogoča. Za takšen primer razvijanja programske opreme je potrebno celoten proces neprekinjene dostave prilagoditi in upoštevati morebitne dodatne zaplete pri izdaji programske opreme v produkcijsko okolje.

1.2 Cilji

V magistrski nalogi bomo predstavili strategije in tehnologije, ki nam omogočajo postavitev takšnega sistema. Slednje bomo opisali na realnih projektih in pripravili orodja za izvajanje integracijskega ter dostavnega procesa. Bolj podrobno bomo opisali tudi proces neprekinjene integracije in dostave v okolje, kjer nimamo dostopa do nobenega izmed programskih okolij in nimamo možnosti odriva programske kode v strankino omrežje.

Poglavje 2

Teoretična osnova

2.1 Poslovno kritične aplikacije

Poslovno kritične aplikacije so aplikacije, ki so nujne za delovanje podjetja. To so tiste aplikacije, ki bi v primeru odpovedi ali prekinitve delovanja močno vplivale na poslovanje podjetja in potencialno povzročile veliko finančne škode ali škodovale ugledu podjetja. Kritičnost aplikacije, ki ga neko podjetje uporablja, je odvisno od podjetja in narave dela, ki ga to podjetje opravlja. Primeri poslovno kritičnih aplikacij so:

- Aplikacije za elektronsko poslovanje: Spletne aplikacije, ki omogočajo podjetjem da tržijo svoje izdelke in storitve preko spleta.

- Aplikacije za upravljanje s strankami (CRM): Aplikacije, ki pomagajo podjetjem upravljati poslovanje s strankami in hranijo podatke o strankah.

- Dobavni sistemi (SCM): Sistemi, ki pomagajo podjetju upravljati z dobavo in prodajo, kot tudi z beleženjem inventarja in logistiko.

- Sistem za načrtovanje virov (ERP): Sistem za nadzor poslovanja podjetja.

- Sistemi poslovne inteligence: To so aplikacije ki pomagajo podjetju zbirati, hraniti in analizirati podatke za sklepanje boljših poslovnih odločitev.

- Orodja za komunikacijo: Aplikacije, ki zaposlenim omogočajo komunikacijo kot na primer: elektronska pošta, sporočilni sistemi in aplikacije za video konference.

Pri razvoju poslovno kritičnih aplikacij so pomembne naslednje lastnosti:

- Zanesljivost: Aplikacija mora biti vedno na voljo in delovati konsistentno brez napak in zaustavitev.

- Skalabilnost (razširljivost?): Ob rasti podjetja se količina prometa, ki jih mora aplikacija obdelati, lahko poveča. Aplikacija mora zato biti sposobna obvladovati velike količine prometa in podpreti rastoče zahteve podjetja.

- Varnost: Poslovno kritične aplikacije pogosto hranijo občutljive podatke, na primer podatke o uporabnikih ali finančne podatke. Aplikacija mora zato biti varna in zaščitena pred kibernetskimi napadi, nepooblaščenim dostopom in drugim tveganji, da prepreči izlive takšnih podatkov.

- Zmogljivost: Počasna ali neodzivna aplikacija negativno vpliva na produktivnosti. Aplikacija mora zato biti odzivna in delovati dobro tudi pod velikimi obremenitvami.

- Vzdrževanje: Dobro vzdrževanje aplikacije je pomembno, da aplikacija ostane zanesljiva in zmogljiva. Aplikacija mora zato biti lahka za vzdrževanje in posodabljanje kot tudi jasno in dobro dokumentirana.

- Uporabniška izkušnja: Dober uporabniški vmesnik in izkušnja lahko uporabnikom olajšata razumevanje in uporabo aplikacije, kar poveča produktivnost.

- Prilagodljivost: Različna podjetja imajo različne potrebe, zato je pomembno, da je aplikacija prožna in prilagodljiva.

2.2 Neprekinjena integracija

Neprekinjena integracija je praksa razvoja programske opreme, pri kateri programerji redno združujejo svoje spremembe kode v centralni repozitorij. Ta koda se nato avtomatsko zgradi, preizkusi in objavi [4]. Cilj neprekinjene integracije je čim prej avtomatsko odkriti napake, tako da lahko programerji posvečajo več časa pisanju kode, kot iskanju napak [5]. Avtomatska gradnja, preizkušanje in objavljanje kode tudi pomaga ekipam, da objavijo posodobitve pogosteje in z večjim zaupanjem, kar je še posebej pomembno v hitro spreminjajočih se okoljih razvoja [5].

Za sistem neprekinjene integracije in dostave so potrebni trije glavni elementi: centralni repozitorij, orodje za avtomatizacijo gradnje in orodje za preizkušanje [**?**, **?**]. Centralni repozitorij je mesto, kamor razvijalci oddajo spremembe svoje kode. Orodje za avtomatizacijo gradnje je odgovorno za avtomatsko gradnjo in objavljanje posodobljene kode. Orodje za preizkušanje pa izvaja avtomatsko testiranje na posodobljeni kodi, s čimer zagotovi da posodobljena koda deluje pravilno in dosega zahtevane standarde kakovosti [5].

2.2.1 Razlogi za neprekinjeno integracijo

Ko razvijalec želi dodati nekaj novega v programsko kodo aplikacije, si v svojem okolju ustvari kopijo temeljne programske kode, ki je v tistem trenutku aktualna. Medtem, ko razvijalec v svojem okolju razvija nove funkcionalnosti ali pripravlja popravke, lahko ostali razvijalci spremenijo temeljno programsko kodo. Zato se lokalna kopija razvijalca čedalje bolj razlikuje od temeljne programske kode. Še več, razvijalci lahko v temeljno programsko kodo dodajo nove funkcionalnosti, nove knjižnice ali druge vire, ki lahko ustvarijo dodatne odvisnosti in potencialne konflikte.

Dalj časa kot razvijalec svoje lokalne kopije ne združi s temeljno programsko kodo, večje je tveganje integracijskih konfliktov in napak pri združevanju kode [6]. Preden razvijalec svoje spremembe doda v glavno vejo, mora najprej posodobiti svojo lokalno kopijo, da pridobi vse spremembe, ki so bile v vmesnem času dodane v glavno vejo. Več kot je bilo sprememb dodanih v vmesnem času, več dela ima razvijalec, preden svoje spremembe doda v glavno vejo.

Če razvijalec predolgo odlaša z oddajo kode v glavno vejo, se njegova kopija lahko začne zelo razlikovati od kode v glavni veji in za integracijo svoje kode v glavno vejo porabi več časa, kot ga je za razvoj sprememb. Temu rečemo tudi "integracijski pekel"[7].

Implementacija neprekinjene integracije lahko prinese številne koristi pri razvoju programske opreme, vključno z izboljšano kakovostjo kode, hitrejšim objavljanjem posodobitev in zmanjšanjem tveganja napak pri integraciji in objavi [?, ?]. Vendar pa implementacija neprekinjene integracije predstavlja tudi nekaj izzivov, kot je potreba po namestitvi in vzdrževanju infrastrukture za neprekinjeno integracijo in potreba po zagotavljanju učinkovitega postopka preizkušanja programske kode [?, ?].

2.3 Neprekinjena dostava

Neprekinjena dostava je metoda razvoja in izdelave, ki temelji na stalnem izboljševanju procesov in izdelkov ter zagotavljanju neprekinjene dostave izdelkov ali storitev. To pomeni, da se procesi in izdelki izboljšujejo in nadgrajujejo neprekinjeno ter dostava izdelkov ali storitev ne povzroči prekinitev ali zastoja v procesu.

Cilj neprekinjene dostave je zmanjšanje časa med pisanjem kode in njene dostave končnim uporabnikom obenem pa zagotavljanje visoke kakovosti in zanesljivosti [8].

Neprekinjena dostava je zato širši pomen, ki vključuje neprekinjeno integracijo in avtomatično testiranje obenem pa tudi izdajanje novih verzij in

avtomatično objavo aplikacije v testno ali produkcijsko okolje. Pomemben del pa je tudi sistem za spremljanje napak in težav z zmogljivostjo, ki razvojni ekipi pošilja povratne informacije o delovanju aplikacije, ki jim pomaga identificirati in odpraviti napake [8].

2.3.1 Dostava izvorne kode

Pomemben del neprekinjene dostave je dostava izvorne kode stranki ali končnim uporabnikom. Proces prenosa kode mora biti dobro zasnovan, da zagotovimo, da se izvorna koda pravilno in učinkovito prenese.

Dostava izvorne kode stranki je lahko koristna tako za stranko kot tudi za podjetje, ki je naredilo izvorno kodo. Za stranko je koristno, ker ni več odvisno od podjetja iz vidika posodobitev, vzdrževanja in podpore, vendar za to lahko najame drugo podjetje ali pa ta del sami prevzamejo. To je lahko zelo koristno predvsem takrat, ko podjetje, ki je ustvarilo izvorno kodo, preneha s svojim poslovanjem ali pa ni zmožno zagotoviti podpore zaradi kakšnega drugega razloga.

Dostava izvorne kode pa je koristna tudi za podjetje, ki je izvorno kodo naredilo, predvsem iz vidika, da podjetje ni več odgovorno za izvorno kodo:

- Prenos odgovornosti: z dostavo izvorne kode stranki lahko podjetje prenese del odgovornosti programske opreme na stranko. To je lahko še posebej koristno, če stranka načrtuje prilagoditve ali spremembe programske opreme, saj podjetje ne bi odgovorno za morebitne težave ali napake, ki bi lahko nastali zaradi teh sprememb [8].

- Zmanjšanje bremena vzdrževanja in podpore: dostava izvorne kode lahko zmanjša breme vzdrževanja in podpore podjetja. Če ima stranka dostop do izvorne kode, lahko sama odpravi težave in jih popravi, namesto da bi se zanašala na podjetje za podporo [8].

2.3.2 Neprekinjena namestitev

Najširši obseg pa predstavlja neprekinjena namestitev, ki je praska razvoja programske opreme, pri kateri se spremembe kode avtomatsko zgradijo, preizkusijo, oddajo naročniku in tudi objavijo v produkcijsko okolje, brez potrebe po človeškem posredovanju [4]. Predstavlja še en korak naprej od neprekinjene dostave, pri kateri se koda zgradi in preizkusi, vendar jih mora človek ročno objaviti v produkcijo [9].

Cilj neprekinjene namestitve je, da se nove funkcionalnosti in popravki čim hitreje dostavijo končnim uporabnikom [9]. Da bi to dosegli, uporabljamo neprekinjeno integracijo, neprekinjeno dostavo in specifične prakse neprekinjene namestitve, kot je proces upravljanja in rezervacije računalniških virov z uporabo datotek, ki jih računalnik zna prebrati [10] - infrastruktura kot koda[11].

Neprekinjena namestitev lahko pomaga podjetju, da pogosteje objavi nove funkcionalnosti in posodobitve [12] in izboljša učinkovitost in hitrost razvojnega procesa aplikacije [4].

2.4 Struktura projekta

Struktura projekta opisuje kako je projekt organiziran in razmerja med različnimi deli projekta. Dobro strukturiran projekt pripomore k boljšemu razumevanju, vzdrževanju in razvijanju programske kode projekta. Obstaja veliko načinov strukturiranja projektov, seveda pa je struktura projekta odvisna od specifičnih potreb in ciljev projekta. Pri strukturiranju projekta so ključni deli:

- Direktorijska struktura: Opisuje kako so datoteke in direktoriji razporejeni na datotečnem sistemu. Dobra direktorijska struktura združuje odvisne datoteke in vire ter olajša iskanje datotek.

- Moduli in odvisnosti: V večini projektov, je programska koda logično razdeljena na več modulov ali komponent, kjer vsak izmed modulov

zadolžen za en del projekta. Ti moduli so lahko tudi odvisni med sabo, zato jih je potrebno skrbno organizirati in poskrbeti da ne pride do cikličnih odvisnosti.

- Skripti za gradnjo in namestitev projekta: Skripti opisujejo kako projekt zgraditi in namestiti na testna in produkcijsko okolje. Ti skripti so po navadi del neprekinjene namestitve in morajo zato biti dobro organizirani in enostavni za razumevanje.

- Dokumentacija: Dobra dokumentacija je pomemben del vsakega projekta. Ta lahko vključuje uporabniška navodila, dokumentacijo aplikacijskega programskega vmesnika (API), in druge dokumente, ki pomagajo razvijalcem pri razvoju in uporabnikom pri razumevanju delovanja projekta.

Na projektno strukturo močno vpliva tudi sistem za verzioniranje kode, ki ga uporabljamo [13], saj lahko kodo shranjujemo v enem skupnem repozitoriju ali pa jo razdelimo v več repozitorijev. Izbira repozitorijske strukture je ključna za strukturo projekta, saj ima vsaka repozitorijska struktura različne prednosti in slabosti [14]. Najbolj pogosti repozitorijski strukturi sta: monorepo in polyrepo.

2.4.1 Monorepo

Monorepo ali monolitni repozitorij je poimenovanje organizacije in verzioniranja izvorne kode z enim repozitorijem. Ta lahko vsebuje več projektov, paketov ali modulov in razvijalcem omogoča širok dostop do izvorne kode, skupnih orodji in skupni množici odvisnosti na enem mestu [15].

Prednosti monolitnega repozitorija so:

- Boljša preglednost izvorne kode: Razvijalci lažje najdejo relevantno dokumentacijo, primere implementacij in uporabe, kar pozitivno vpliva na hitrost in kvaliteto kode [15, 16].

- Enostavne odvisnosti: Vsi paketi in moduli v repozitoriju imajo lahko enako verzijo in ni potrebno skrbeti katere verzije so med seboj kompatibilne.

- Lažje spreminjanje odvisnih delov kode: Pri popravljanju in posodabljanju kode lahko popravimo tudi vse dele kode, ki so odvisni od spremenjene kode in vse skupaj oddamo v repozitorij kot eno spremembo.

Slabosti monolitnega repozitorija so:

- Velikost repozitorija: Monolitni repozitorij lahko skozi razvoj zasede veliko prostora na podatkovnem sistemu. To lahko oteži prenos izvorne kode iz repozitorija in ostale operacije sistema za verzioniranje [17].

- Kompleksen cevovod za neprekinjeno dostavo: Kompleksnost konfiguracije cevovoda za neprekinjeno dostavo se poveča, saj je potrebno vse korake izvesti na vseh modulih in projektih. Da ohranimo učinkovitost, pa nočemo vedno izvajati vseh korakov za vse module, saj to podaljša izvajanje cevovoda.

- Veliki zahtevki za združitev vej: Ker repozitorij vsebuje vse odvisne dele kode, je ob posodobitvah lahko posodobljenih veliko vrstic izvorne kode. To lahko povzroči slabo preglednost sprememb v pregledu sprememb zahtevka za združitev [17].

- Orodja za upravljanje: Za učinkovito upravljanje monolitnega repozitorija so velikokrat potrebna dodatna programska orodja kot so orodja za gradnjo aplikacije in upravljalci paketov. To lahko poveča kompleksnost razvijanja aplikacije.

2.4.2 Polirepo

Polirepo je poimenovanje organizacije in verzioniranja izvorne kode z več repozitoriji. Ti vsebujejo vse pakete in komponente izvorne kode, ki skupaj tvorijo celotno aplikacijo [16].

Prednosti uporabe več repozitorijev:

- Poenostavljeno upravljanje: S posameznimi moduli ali paketi v ločenih repozitorijih je lažje slediti spremembam in ugotoviti, katere spremembe spadajo k posameznim projektom.
- Ločitev odgovornosti: Shranjevanje posameznih projektov v ločenih repozitorijih lahko pomaga izolirati kodo in zmanjša tveganje za nastanek sporov med različnimi deli projekta.
- Fleksibilnost: S posameznimi projekti v ločenih repozitorijih je lažje, da se različne ekipe ali razvijalci lahko posvečajo različnim delom projekta brez motenj od drugih ekip.
- Velikost repozitorijev: Ker je celoten projekt razdeljen na več repozitorijev, so ti repozitoriji manjši in zato lažji za prenos in delo na lokalnih računalnikih, še posebej, če razvijalec dela samo na enem delu projekta.
- Potencialno hitrejše gradnje: S posameznimi moduli ali paketi v ločenih repozitorijih jih je morda mogoče graditi neodvisno, kar lahko skrajša skupni čas gradnje.

Slabosti uporabe več repozitorijev:

- Verzioniranje in določevanje odvisnosti: Potrebno je določiti kako se bodo moduli ali paketi v posameznih repozitorijih verzionirali in kako bodo določene odvisnosti med njimi.
- Oteženo spreminjanje odvisnih delov kode: Po spremembi nekega dela kode, se velikokrat zgodi, da je potrebno popraviti še kodo v odvisnih modulih ali paketih. To pomeni, da je treba v vsakem izmed odvisnih repozitorijev dodati spremembo in po možnosti popraviti verzijo.

Oba pristopa imata prednosti in slabosti, in tisto, kar en razvijalec šteje za prednost, lahko drugemu predstavlja slabost. Na primer, nekatere osebe lahko vidijo enostavnost upravljanja enega samega repozitorija kot prednost

uporabe monolitnega repozitorija, medtem ko druge lahko zaradi potencialne povečane zapletenosti vidijo to kot slabost. Podobno lahko nekatere osebe vidijo sposobnost enostavnega deljenja kode in virov med projekti kot prednost uporabe več repozitorijev, medtem ko druge lahko vidijo potrebo po upravljanju večih repozitorijev kot slabost. Zato je izbira med uporabo monolitnega repozitorija ali več repozitorijev največkrat stvar osebnega okusa in je odvisna od specifičnih potreb ter narave projekta kot tudi delovnih navad podjetja.

Poglavje 3

Uporabljene tehnologije in aplikacije

3.1 Repozitoriji programske opreme

3.1.1 Nexus

3.1.2 Git

3.1.3 Register slik Docker

3.2 Gitlab

3.2.1 Neprekinjena integracija in dostava v Gitlab-u

3.2.2 Vzorci gitlab

3.3 Kubernetes

3.4 Konfiguracijski strežniki

3.4.1 Consul

3.4.2 konfiguracijski strežnik Spring Boot

3.5 Zagotavljanje varnosti

3.5.1 Kubernetes v izoliranem produkcijskem okolju

3.5.2 Jsonnet

Poglavje 4

Implementacija

4.1 Potek integracije in dostave

4.1.1 Proces v podjetju

4.1.2 Proces pri stranki

4.1.3 Proces neprekinjene dostave stranki

4.2 Razvoj komponente za neprekinjeno dostavo in integracijsko testiranje - Medius CD

4.3 Postavitev projekta

4.3.1 Java in Maven

4.3.2 Java in Gradle

4.3.3 JavaScript in NPM

Poglavje 5

Rezultati

5.1 Predstavitev projekta

5.2 Primer uporabe na projektu

5.2.1 Java projekt

5.2.2 JavaScript projekt

5.2.3 Python projekt

5.2.4 Monorepo projekt

Poglavje 6

Zaključek

Literatura

[1] licence-cc.pdf.
URL https://ucilnica.fri.uni-lj.si/course/view.php?id=274

[2] M. Hinchey, L. Coyle, Evolving Critical Systems: A Research Agenda for Computer-Based Systems, in: 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, IEEE, 2010, pp. 430–435.

[3] J. Snygg, A. Smedberg, G. Juell-Skielse, Problem management of business-critical systems in literature: a review and viable systems analysis, International Journal of Business Continuity and Risk Management 6 (2016) 238.

[4]

[5] M. Fowler, Continuous integration (2006).
URL https://martinfowler.com/articles/continuousIntegration.html

[6] P. Duvall, S. Matyas, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, 1st Edition, Addison-Wesley Professional, 2007.

[7] W. Cunningham, Integration hell (2006).
URL http://wiki.c2.com/?IntegrationHell

[8] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1st Edition, Addison-Wesley Professional, 2010.

[9]

[10] A. Wittig, M. Wittig, Amazon Web Services in Action, Manning Press, 2016.

[11] J. Humble, D. Farley, Continuous delivery: Reliable software releases through build, test, and deployment automation, Addison-Wesley Professional, 2014.

[12]

[13] Architectural pattern (2023).
URL https://en.wikipedia.org/wiki/Architectural_pattern

[14] The impact of the software architecture on the developer productivity, Pollack Periodica 17 (2022) 7–11.

[15] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, E. Murphy-Hill, Advantages and disadvantages of a monolithic repository: A case study at google, IEEE Computer Society, 2018, pp. 225–234. doi:10.1145/3183519.3183550.

[16] Comparison between mono and multi repository structures, Pollack Periodica 17 (2022) 7–12. doi:10.1556/606.2022.00526.

[17] B. Harry, The largest git repo on the planet (2017).
URL https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/