Zbornik radova Fakulteta tehničkih nauka, Novi Sad

**UDK: 007.5**
**DOI: https://doi.org/10.24867/17IH01Ivic**

# JEDNO REŠENJE OKRUŽENJA ZA ISPITIVANJE AUTOMOTIV PLATFORME U KONTINUALNOJ INTEGRACIJI

# ONE SOLUTION OF ENVIRONMENT FOR EXAMINING AUTOMOTIVE PLATFORM IN CONTINUOUS INTEGRATION

Irena Ivić, Vladimir Rajs, *Fakultet tehničkih nauka, Novi Sad*

**Oblast - MEHATRONIKA**

**Kratak sadržaj –** *Tema ovog rada je analiza rešenja za automatizaciju procesa koji je potrebno pokretati na dnevnom nivou, a koji je pre datog rešenja, trebalo izvršavati manuelno.*

**Ključne reči:** *automatizacija, kontinualana integracija, automotiv, Jenkins*

**Abstract** – *This paper analyzes solution for automatization of process which needs to be executed daily, and which, before the given solution, needed to be executed manually.*

**Keywords:** *automatization, continuous integration, automotive, Jenkins*

## 1. UVOD

Tokom razvoja softvera, trebalo bi voditi računa o tome da se, između ostalog, smanji utrošak vremena na procese koji bi mogli da se automatizuju. Automatizacijom procesa uklanja se mogućnost nastajanja čovekove greške, i razvoj softvera je brži s obzirom da nije potrebno manuelno pokretati uvek iste procese. U radu je pomoću programskog jezika Pajton i alata za kontinualnu integraciju *Jenkins*, automatizovan proces koji je potrebno pokretati na dnevnom nivou, a koji je pre datog rešenja, trebalo izvršavati manuelno. Kontinualna integracija (engl. *CI – Continuous Integration*) je praksa u softverskom inžinjeringu u kojoj se teži ka tome da male izmene u kodu budu integrisane u repozitorijum, u cilju ranog otkrivanja grešaka i bržeg razvoja, kao i da se manje vremena troši na *debug* – ovanje pa ostaje više vremena za razvoj. *CI/CD* predstavlja skup paradigmi kontinualne integracije i kontinalne isporuke koji je našao široku primenu u industriji. *CI* predstavlja pristup u kome se programeri podstiču da implementiraju male izmene na kodu što je frekventnije moguće. Dalje, nema čekanja da se sazna da li kod radi, i smanjuju se problemi integracije.

## 2. SISTEM ZA KONTROLU VERZIJE

Sistem za kontrolu verzije (engl. *VCS - Version Control System*) upravlja promenama nad fajlovima ili direktorijumom. Sistemom za kontrolu verzije pamte se promene nastale tokom vremena i time se dozvoljava vraćanje fajlova ili projekta na prethodnu verziju, poređenje izmena tokom vremena, pregled poslednje izmene da bi se videlo šta bi moglo da bude uzrok nastalog problema, itd. Da bi se odredilo na kojem direktorijumu ili skupu fajlova treba da se prate promene, potrebno je preuzimanje repozitorijuma sa *host* mašine. Svaka od promena praćena je automatski. Umesto čuvanja svake promene posebno, *VCS* sačeka da se promene podnesu kao kolekcija više akcija, koja se naziva *commit*. Proces praćenja promena je pregledan sve dok se ne *commit* – uju ove promene [4].

### 2.1 *Git*

*Git* predstavlja distribuirani sistem za kontrolu verzije (engl. *DVCS - Distributed Version Control System*) kojim se prate promene u fajlovima, i koordinira između tih fajlova. Kod *DVCS* sistema, klijenti ne preuzimaju samo trenutan izgled fajlova, već se preslikava ceo repozitorijum. Ako neki od servera prestane sa radom, a ovi sistemi su povezani pomoću tog servera, svaki od klijentovih repozitorijuma može da se iskopira nazad na server da bi se obnovio [3].

## 3. KONTINUALNA INTEGRACIJA

Kontinualna integracija (engl. *CI – Continuous Integration*) je praksa u softverskom inžinjeringu u kojoj se teži ka tome da male izmene u kodu budu integrisane u repozitorijum, u cilju ranog otkrivanja grešaka i bržeg razvoja. *CI* predstavlja automatizaciju promena koda prilikom integracije više projekata u jedan softverski projekat. Automatizovani alati se koriste za utvrđivanje ispravnosti novog koda pre integracije. Sistem za kontrolu verzije izvornog koda je suština ovog procesa. Sistem za kontrolu verzije takođe ima druge provere, kao što su automatski testovi za proveru kvaliteta koda. U nastavku će biti opisano funkcionisanje procesa kontinualne integracije. Programeri vrše izmene u lokalnom okruženju. Kada završe, *commit* - uju kod u repozitorijum. *CI* server prati repozitorijum i povlače se nove izmene kada se dogode. *CI* server pokreće *build* i izvršava *unit* testove i testove integracije. *CI* server pravi instancu projekta spremnu za testiranje, i obaveštava tim o uspešnosti *build* - a. Ukoliko je *build* neuspešan, tim odmah može da počne sa rešavanjem problema. Ciklus kontinualne integracije se nastavlja, a projekat se testira. Zatim se pokreće *build*, izvršavaju se testovi i ukoliko postoji neka greška, to se saznaje veoma rano i može odmah da se ispravi.

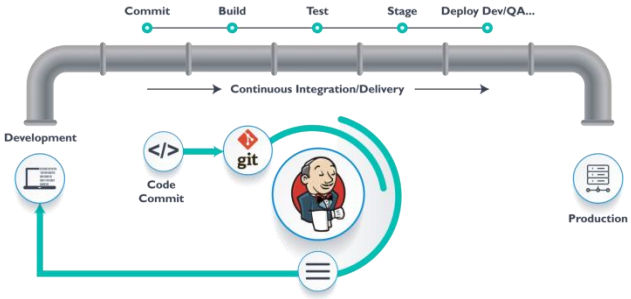
### 3.1 *Jenkins*

Jenkins predstavlja alat za kontinualnu integraciju i kontinualnu isporuku (engl. *CD - Continuous Delivery*).

---

**NAPOMENA:**
**Ovaj rad proistekao je iz master rada čiji mentor je bio prof. dr Vladimir Rajs.**

*CD* znači da je kod spreman za objavljivanje na produkciju u bilo kom trenutku. Originalna verzija *Jenkins* - a je fleksibilna, i to predstavlja jednu od velikih prednosti jer se time dobija mogućnost da se ovaj alat primeni u mnogim slučajevima.




Slika 1. *Jenkins*

Postoji veliki broj *Jenkins plugin* opcija kojim se dobijaju dodatne funkcionalnosti, dozvoljavajući integraciju dodatnih alata, uključujući *Gradle*, *Groovy*, *Maven*, itd. Iako se *Jenkins* upotrebljava u mnoge svrhe, proces uglavnom izgleda fundamentalno slično:

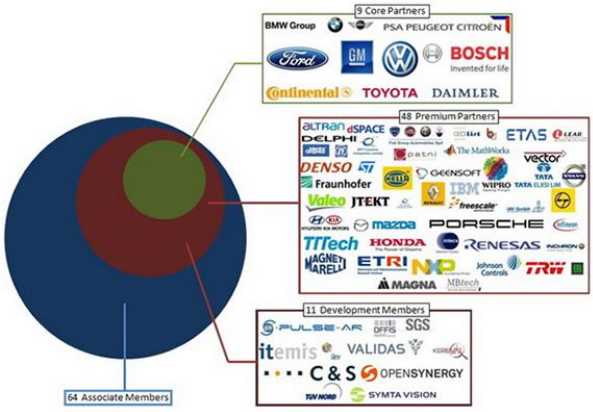
- Developeri *commit* – uju kod do repozitorijuma izvornog koda
- *Jenkins* proverava repozitorijum redovno da vidi da li je došlo do nekih izmena
- Kad *Jenkins* registruje promene, odmah kompajlira kod
- Ako postoji neuspešan *build*, poruka koja sadrži grešku biće poslata developer
- Ako kod nema grešaka, biće isporučen na produkciju.

S obzirom da se izmene na kodu rade tokom dana, *build* – ovanje se pokreće noću i bitno pravilo tokom razvoja softvera je da se to *build* – ovanje ne prekida. Pre *Jenkins* - a, developeri, da bi izbegli prekidanje noćnog *build* - a, *build* – ovali i testirali su kod na lokalnoj mašini pre *commit* – ovanja koda. To bi značilo proveravati promene jednog dela koda, bez dnevnih *commit* – ovanja drugih delova koda. Osim mehanizama za automatski build i testiranje, *Jenkins* se može podesiti tako da automatizuje još poslova. Na primer, ako je *build* uspešan (svi testovi su uspešni), kod se može dopremiti na produkciju. Ova praksa se naziva *Continuous Deployment* i često je usko povezana sa *CI* -om [2].

## 4. *AUTOSAR* STANDARD

*AUTOSAR* (engl. *AUTomotive Open System Architecture)* grupa osnovana je 2013. godine i danas je čine proizvođači automobila, automobilske opreme, proizvođači alata i proizvođači poluprovodnika. *AUTOSAR* je najpre objavljen kao standard *AUTOSAR Classic Platform* za namenske elektronske kontrolne jedinice (engl. *ECU - Electronic Control Unit*) u realnom vremenu zasnovanom na *OSEK* operativnom sistemu, a potom kao standard *AA Platform* radi novonastale potrebe za dinamičkom softverskom arhitekturom u vozilu. *AUTOSAR* grupa je razvila standard koji je jedan od vodećih u automobilskoj industriji za razvoj softvera.
Cilj ovog standarda jeste da pruži skup specifikacija kojim se opisuju osnovni softverski moduli, definišu programske sprege i realizuju zajedničke metode daljeg razvijanja na osnovu standardizovanog formata. Najznačajnija osobina ovog standarda je ta što se on može koristiti u vozilima različitih proizvođača, ali mogu ga koristiti i različiti proizvođači elektronske opreme koja se koristi u vozilima. Ova osobina znatno smanjuje troškove i vreme koje bi bilo utrošeno da kompanije samostalno rade na nekom sličnom softveru, pogotovo što kompleksnost vozila sve više raste [1]. AUTOSAR čini troslojna arhitektura:

- Osnovni softver *(*engl. *Basic Software*) – standardizovani softverski moduli koji su neophodni za funkcionisanje višeg softverskog sloja.
- Izvršno okruženje (engl. *RTE – runtime environment*) – posrednički softver (engl. *middleware*) koja realizuje komunikaciju softverskih komponenti i osnovnog softvera.
- Aplikativni sloj (engl. *Application Layer*) – komponente aplikativnog softvera koje komuniciraju sa *RTE*.




Slika 2. *AUTOSAR* grupa

## 5. KONCEPT REŠENJA

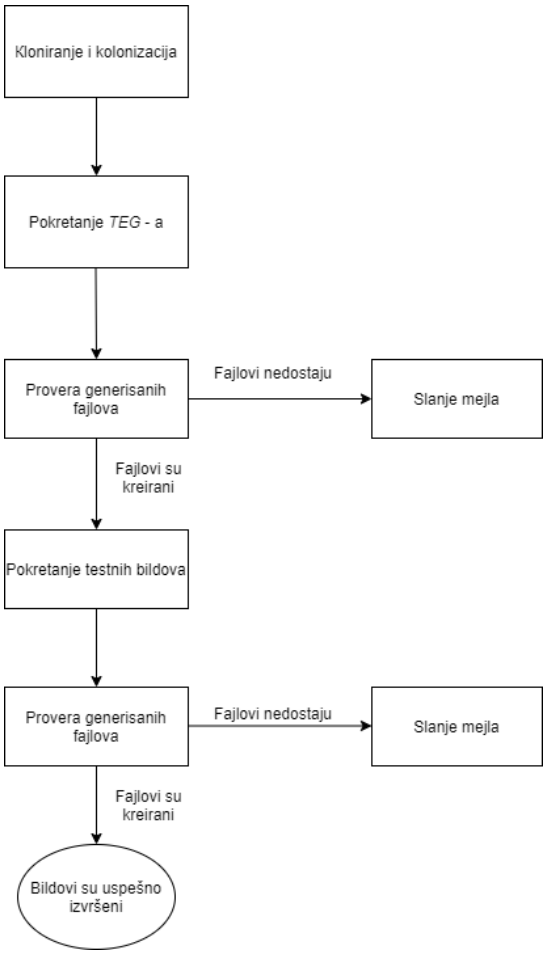
Ovo poglavlje je podeljeno u četiri potpoglavlja sledećih naziva: analiza problema, algoritam za rešenje problema, realizacija rešenja i problemi pri realizaciji. U prvom potpoglavlju, analiza problema, dat je opis problema koji je potrebno rešiti. U drugom potpoglavlju predstavljen je algoritam realizacije datog problema. U trećem potpoglavlju, detaljnije su objašnjeni koraci rešavanja, dok su u četvrtom potpoglavlju predstavljeni problemi koji su se pojavili tokom realizacije.

### 5.1. Analiza problema

Prilikom razvoja softverskog rešenja potrebno je automatizovati proces pokretanja testnih bildova, koji se nakon izvršavanja spuštaju na hardver na kojem se proverava njihova funkcionalnost. Proces je potrebno, ukoliko je moguće, napraviti što više generičkim da bi mogao da se primeni na veći broj projekata. Prilikom automatizacije ovog procesa potrebno je slediti nekoliko koraka koji su se u prethodnim verzijama rešavanja ovog problema, pre automatizovanja, radili manuelno. Neophodno je pripremiti radno okruženje, pokrenuti interni alat *TEG* kojim se generišu fajlovi neophodni za pokretanje testnih bildova, i nakon toga izvršiti testne bildove.

Za svaki od ovih koraka potrebno je proveriti da li je uspešno realizovan. Prednost automatizacije procesa je ta što tokom rada doprinosi uklanjanju mogućnosti nastajanja greške uticajem čoveka, smanjenju utroška vremena na postupke koji su ponavljajući, standardizaciji nekog postupka, brzini rešavanja nekog problema, itd.

### 5.2 Algoritam rešenja problema




Slika 3. Algoritam

### 5.3 Realizacija rešenja

Rešenje je realizovano u *Jenkins* – u, čiji će interfejs, kao i funkcionalnosti koje poseduje biti objašnjene u nastavku. *Jenkins Pipeline* sadrži mnogo plugin - ova koji omogućavaju implementaciju i integraciju *continuous delivery pipeline* - ova u *Jenkins* - u.
Od verzije 2.5. *plugin* – a za pipeline, postoje dva tipa *pipeline* – a, deklarativni i skriptovani.
Većinu funkcionalnosti koje omogućava *Groovy* jezik moguće je koristiti u skriptovanom *pipeline* - u. Najvažniji aspekt ovog *pipeline* – a je kontrola toka.
Skriptovani *pipeline* obezbeđuje mali broj ograničenja definisana u samom *Groovy* jeziku, ne u *pipeline* - u, što ga čini idealnim izborom za iskusnije korisnike i korisnike koji imaju kompleksne zahteve.
Deklarativni pipeline promoviše deklarativni model programiranja, dok skriptovani pipeline promoviše imperativni. U ovom radu je korišćen skriptovani pipeline, i stoga će njegove osnove biti date u nastavku [2].

U skriptovanoj pipeline sintaksi, jedan ili više *node* blokova izvršavaju glavni deo tokom čitavog *pipeline* – a. Iako ovo nije obavezan deo skriptovanog *pipeline* – a, ograničavanje *pipeline* posla unutar *node* blokova daje dve stvari:

- Raspoređuje korake unutar bloka koje treba da pokrene dodavajući ih u red, i čim je *node* slobodan za pokretanje *Jenkins job* – a, izvršiće se tim redosledom.
- Kreira *workspace* (radni direktorijum specifičan za taj određeni *pipeline*) gde posao može da se izvrši.

Softversko rešenje ovog problema dato je pomoću skriptovanog *pipeline* – a, *node* predstavlja lokalni računar, i rešenje je podeljeno u pet *stage* – eva. U rešenju, dati su sledeći koraci:

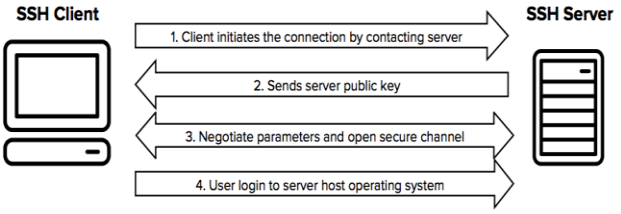
- Definisanje *'Cloning and colonization' stage* – a
- Izvršavanje koraka unutar *'Cloning and colonization' stage* – a
- Definisanje *'Running TEG' stage* – a
- Izvršavanje koraka unutar *'Running TEG' stage* – a
- Definisanje *'Checking output of TEG' stage* – a
- Izvršavanje koraka unutar *'Checking output of TEG' stage* – a
- Definisanje *'Running build' stage* – a
- Izvršavanje koraka unutar *'Running build' stage* – a
- Definisanje *'Checking output of build' stage* – a
- Izvršavanje koraka unutar *'Checking output of build ' stage* – a

### 5.4 Problemi pri realizaciji

U nastavku će biti opisana dva problema pri realizaciji: autentifikacija i zaštita od grešaka.

#### 5.4.1 Autentifikacija

Prilikom kloniranja projekta sa *Git* - a dostupno je nekoliko protokola: *SSH, GIT i HTTP/ HTTPS*. Potrebno je izabrati protkol kojim će se obezbediti bezbedan prenos podataka.
SSH (engl. *Secure Shell*) je mrežni protokol koji korisnicima omogućava uspostavljanje sigurnog komunikacijskog kanala između dva računara putem nesigurne računarske mreže. S obzirom da je *SSH* autentifikovani protokol, potrebno je uspostaviti akreditive sa serverom pre povezivanja. Autentifikacija je urađena pomoću javnog ključa [5]. Prednost *SSH* protokola u odnosu na *HTTPS* je korišćenje ključa jer je to sigurnije u odnosu na šifru. S obzirom da nemamo šifru za *SSH*, to ne zahteva dvofaktorsku autentifikaciju kao što je to slučaj sa *HTTPS*. Za svaku akciju koja se koristi u *Git* – u, *SSH* uklanja teret autentifikacije na *remote* serveru i to predstavlja jednu od glavnih prednosti. Ko god ima potreban privatni ključ može da *push* - uje na repozitorijum bez potreba za uređajem za generisanje koda. Ako se desi da je privatni ključ ukraden, neko može da *push* - uje na nove prazne repozitorijume i ukloni sve zapise promena i istorije za svaki repozitorijum koji se tu nalazi, ali ne može da se promene ništa na *GitHub* nalogu. Iz navedenih razloga, korišćen protokol u ovom zadatku je *SSH*.




Slika 4. *SSH* konekcija

### 5.4.2 Zaštita koda od grešaka

Prilikom razvijanja softvera, potrebno je voditi računa o slučajevima tokom kojih može da dođe do prekida izvršavanja koda, i načinama na koji se kod može zaštiti od potencijalnih grešaka.

Greška može da ukazuje na kritične probleme koje program ne bi trebalo da pokuša da uhvati, ali takođe postoje i izuzeci koji mogu da ukazuju na uslove koje aplikacija treba da pokuša da uhvati. Greške su oblik neobrađenih izuzetaka i nepopravljive su, prekida se izvršavanje programa, i programer ne bi trebalo da pokuša da ih obradi, dok izuzeci nisu greške zbog kojih je neophodno prekinuti izvršavanje programa. Izuzetke je moguće predvideti, kao i rukovati njima tokom izvršavanja programa.

Postoje ugrađeni izuzeci koji su obrađeni od strane programa, a moguće je napisati i korisnički definisane izuzetke, i tada je potrebno naslediti klasu *Exception*. Postoje različiti tipovi izuzetaka, a kao deo poruke o grešci uvek je ispisano o kom tipu izuzetka se radi. Izuzecima se u Pajtonu rukuje pomoću *try* - *except* klauzule, gde se u okviru *try* bloka piše deo koda u kojem može doći do nekog tipa izuzetka, a nakon ključne reči *except* navode se tipovi izuzetaka kojima se rukuje, a zatim i način na koji se to može uraditi. U ovom radu, kod je zaštićen od potencijalnog pada primenom izuzetaka. Obrada izuzetaka je mehanizam koji služi za obradu grešaka, pruža dovoljno informacija o nastaloj grešci, omogućava da se za svaki tip greške kreira odgovarajući način obrade, omogućava odvajanje logike programa od koda kojim se obrađuju greške. Ako se u programskom kodu ne predvide izuzeci - program izbacuje "grubu" poruku o grešci i zaustavlja izvršavanje aplikacije.

## 6. TESTIRANJE

U ovom poglavlju navedeno je pet testova, pomoću kojih se proverava uspešnost realizovanja datog problema. Testovi kao rezultat vraćaju *PASS* ili *FAIL*.

| Rd. Br. | Naziv testa | Rezultat |
|---|---|---|
| 1. | Provera uspešnosti kolonizacije | *PASS* |
| 2. | Izbrisan ulazni modul *TEG* - a | *PASS* |
| 3. | Provera uspešnosti pokretanja *TEG* - a | *PASS* |
| 4. | Izbrisan ulaz *build* - a | *PASS* |
| 5. | Provera uspešnosti pokretanja *build* - a | *PASS* |

Tabela 1. Testovi

## 7. ZAKLJUČAK

U ovom radu dat je primer rešenja automatizacije procesa automotiv platforme u kontinualnoj integraciji. Date su teorijske osnove kontinualne integracije, kao i alata *Jenkins*. Objašnjeno je šta je sistem za kontrolu verzije, i date su osnove Git – a. S obzirom da se automatizovao proces automotiv platforme, prikazan je *AUTOSAR* sistem, kao i njegovo funkcionisanje. Takođe, dat je algoritam rešenja i objašnjene su glavne celine rešenja. U okviru rada, urađeno je i testiranje funkcionalnosti softvera, i predstavljeni su testovi koji su se izvršavali.

Pravci za dalje istraživanje i razvoj alata mogli bi da budu sledeći:

- Moguće je integrisati *TestExecutor* alat u *Jenkins job*. *TestExecutor* je interni alat za izvršavanje testova na hardveru. Potrebno je izvršiti postojeći *Jenkins job* za *TestExecutor* ukoliko je postojeći *build* uspešno izvršen.
- Moguće je parametrizovati dati projekat, tako da može da se primeni na slične vrste zahteva koje je potrebno ostvariti.

## 8. LITERATURA

[1] https://www.autosar.org/standards, pristupljeno u septembru 2021.
[2] https://jenkins.io/doc, pristupljeno u septembru 2021.
[3] https://www.freecodecamp.org/news/learn-the-basics-of-git-in-under-10-minutes-da548267cc91/, pristupljeno u septembru 2021.
[4] https://www.atlassian.com/git/tutorials/what-is-version-control, pristupljeno u septembru 2021.
[5] https://www.ssh.com/academy/ssh, pristupljeno u septembru 2021.

**Kratka biografija:**

**Irena Ivić** rođena je u Novom Sadu 1996. god. Srednje obrazovanje stekla je u Gimnaziji “Isidora Sekulić” na prirodno-matematičkom smeru. Osnovne akademske studije iz oblasti Mehatronika, robotika i automatizacija upisala 2015. godine, a diplomirala 2019. Iste godine upisuje master akademske studije, smer “Mehatronika, robotika i automatizacija”.

kontakt: ivicirena96@gmail.com