Kontinuirano testiranje i metrike u DevOps ekosistemu

Uvod i problem jaza između razvoja i produkcije (Slajdovi 1–4)

[Slajd 1]

Danas obrađujemo DevOps ekosistem i bavimo se temom **Continuous Testing-a i metrika**. Naš cilj danas je da pokažemo kako se programersko okruženje direktno povezuje sa produkcijom, i kako da osiguramo vrhunski kvalitet u svetu gde se softver isporučuje izuzetno brzo — kroz praktične primere u Javi, Javalinu i Maven-u.

[Slajd 2]

Naša današnja agenda sastoji se od pet celina. Prvo ćemo sagledati zašto tradicionalno testiranje više ne radi u modernom DevOps okruženju. Zatim ćemo razmotriti Shift-Left i Shift-Right strategije, nakon čega prelazimo na kapije kvaliteta i optimizaciju vremena izvršavanja testova. Na kraju ćemo sve to povezati sa čuvenim DORA metrikama i kompletnim CI/CD pajplajnom.

[Slajd 3]

Krenimo od samog problema. U tradicionalnom razvoju softvera imali smo jasno razdvojene faze: programeri bi nedeljama pisali kod, a onda bi ga prebacili QA timu, koji bi ga testirao nedeljama i tek onda puštao u rad. U modernom DevOps-u kod se isporučuje više puta dnevno. **DevOps** je skup praksi, alata i kultura rada koji spaja razvoj softvera (Development) i IT operacije (Operations) kako bi se aplikacije pravile, testirale i objavljivale brže i pouzdanije. Ako ručno testiranje traje pet dana, ono postaje usko grlo. Rešenje je **Kontinualno Testiranje** — ugrađivanje automatizovanih provera u sam tok isporuke softvera.

[Slajd 4]

Šta je tačno **Kontinualno testiranje**? To je proces izvršavanja automatizovanih testova u svakoj fazi pajplajna isporuke, sa jednim glavnim ciljem: da dobijemo povratnu informaciju o poslovnim rizicima u najkraćem mogućem roku. Da bi to radilo, testovi moraju biti potpuno automatizovani, stabilni i ponovljivi od prve linije koda pa sve do produkcije.

Shift-Left strategija i praksa (Slajdovi 5–13)

[Slajd 5]

Da bismo pokrili ceo životni ciklus, koristimo dve komplementarne strategije: **Shift-Left** i **Shift-Right**. Tradicionalno se testiralo tek na sredini ili kraju procesa. Shift-Left pomera testiranje maksimalno ulevo — na sam početak razvoja, dok Shift-Right pomera određene provere udesno — u samu produkciju. Nijedan od ovih pristupa nije dovoljan sam za sebe; tek zajedno čine kompletan Continuous Testing.

[Slajd 6]

Suština Shift-Left strategije je jednostavna: pronađi bag tamo gde je i nastao — na stolu programera. Što se greška kasnije otkrije, njeno ispravljanje je eksponencijalno skuplje. U

tome nam pomažu tri ključne tehnike: statička analiza koda, Test-Driven Development i provera pre spajanja koda.

[Slajd 7]

Prvi moćan alat za Shift-Left je **statička analiza koda**. To je provera izvornog koda bez njegovog pokretanja. Alati poput Checkstyle-a, SpotBugs-a ili SonarQube-a u stanju su da prepoznaju loše stilske navike, potencijalne bagove i bezbednosne propuste pre nego što se testovi uopšte i kompajliraju.

[Slajd 8]

U Maven-u ovo automatizujemo za svega minut. U naš `pom.xml` dodamo `spotbugs-maven-plugin` i vežemo ga za `verify` fazu build-a. Od tog trenutka, svaki put kada pokrenemo build, SpotBugs automatski skenira prevedeni bajtkod kako bi automatski otkrio potencijalne greške, bezbednosne propuste i loše prakse u programiranju pre pokretanja same aplikacije.

[Slajd 9]

Pogledajte konkretan primer baga koji SpotBugs odmah hvata. U metodi `checkAdmin` programer poredi stringove sa `==` umesto sa `.equals()`, i pritom rizikuje `NullPointerException` ako je `role` null. Čim pokrenemo `mvn clean verify`, SpotBugs zaustavlja build i sprečava da ovakav kod ikada ode dalje.

[Slajd 10]

Drugi stub Shift-Left pristupa je **TDD — Test-Driven Development**, gde testove pišemo pre samog produkcionog koda. Proces se vrti u poznatom ciklusu **Crveno – Zeleno – Refaktoriši**: prvo napišemo test koji pada jer funkcionalnost još ne postoji; zatim napišemo najjednostavniji kod da test prođe; i na kraju sredimo kod bez menjanja ponašanja.

[Slajd 11]

Hajde da vidimo kako TDD izgleda na jednom malom Javalin REST API zadatku. Treba da napravimo endpoint `/greet` koji prima ime i vraća pozdravnu poruku. U `pom.xml` imamo Javalin i JUnit 5.

[Slajd 12]

U prvom koraku — faza **Crveno** — pišemo test `GreetControllerTest` koji poziva `GreetService` i očekuje poruku `"Hello, DevOps!"`. Ovaj kod se trenutno čak i ne kompajlira jer klasa `GreetService` još ne postoji.

[Slajd 13]

Zatim prelazimo u fazu **Zeleno**: kreiramo klasu `GreetService`, implementiramo metodu `createMessage` i povezujemo je sa Javalin rutom `/greet`. Test sada prolazi, a mi smo sigurni da imamo pokrivenu funkcionalnost od samog starta.

Shift-Right strategija i praksa (Slajdovi 14–18)

[Slajd 14]

Sada idemo na drugu stranu vremenske linije — **Shift-Right**. Zašto uopšte testiramo u produkciji? Zato što određene stvari, poput stvarnog mrežnog kašnjenja ili ponašanja baze

pod udarom hiljada korisnika, jednostavno ne možemo verno da repliciramo u testnom okruženju.

[Slajd 15]

Prva ključna tehnika ovde su **Feature Flags**, odnosno zastavice funkcionalnosti. One nam omogućavaju da novu funkcionalnost pošaljemo u produkciju, ali da je držimo ugašenom dok ne budemo spremni. Time razdvajamo tehnički ***deployment*** koda od poslovnog ***release***-a za korisnike. Jedini oprez ovde je tehnički dug — stare zastavice se moraju redovno čistiti iz koda.

[Slajd 16]

U Javi se jednostavan Feature Flag može napraviti kroz pomoćni servis koji čita konfiguraciju. U Javalin kontroleru proveravamo da li je zastavica uključena: ako jeste, korisniku šaljemo novi pozdrav, a ako nije, aplikacija bezbedno izvršava stari kod. Ako nešto pođe po zlu u produkciji, novu funkcionalnost gasimo za sekund, bez novog deployment-a.

[Slajd 17]

Pored zastavica, tu su i "kanarinske isporuke" na engleskom **Canary Deployments,** nekad prevodimo i kao postepeno uvođenje i **A/B testiranje**. Naziv kanarinske isporuke potiče od nekadašnje prakse u rudarstvu da se u okna nosi kanarinac u kavezu. Kanarinci su osetljivi na otrovne gasove pa su služili kao rano upozorenje na njihovo prisustvo. Kod Canary isporuke novu verziju pustimo samo za, recimo, 5% korisnika i pratimo greške — ako sve radi, širimo je na ostale. Sa druge strane, A/B testiranje je metoda dve verzije neke aplikacije (verzije A i verzije B) kako bi se utvrdilo, na primer, koja verzija sajta donosi više kupovina.

[Slajd 18]

Naravno, Shift-Right testiranje je nemoguće bez dobre telemetrije — ako nemamo metrike, u produkciji smo slepi. Pomoću biblioteke **Micrometer** u Javalin lako ugrađujemo brojač HTTP zahteva i izlažemo ga na `/metrics` ruti, odakle ga alati poput Prometheusa mogu pratiti u realnom vremenu.

Kapije kvaliteta i optimizacija testova (Slajdovi 19–27)

[Slajd 19]

Da bismo sve ove provere učinili obaveznim, uvodimo **Kapije kvaliteta (Quality Gates)**. To je automatski set uslova koje kod mora da ispuni da bi prešao u sledeću fazu pajplajna. Ako pokrivenost koda padne ispod 80% ili se pojavi kritična ranjivost, kapija se zatvara bez potrebe da čovek ručno pregleda kod.

[Slajd 20]

Jedna od najčešćih metrika na kapijama kvaliteta je pokrivenost koda (**Code Coverage)**, koju u Javi merimo alatom **JaCoCo**. Ovde uvek naglašavam jednu važnu stvar: 100% pokrivenosti ne znači da su testovi dobri, jer možete izvršiti kod bez ikakve provere rezultata. Ali niska pokrivenost je siguran dokaz da deo koda uopšte nije testiran.

[Slajd 21–22]

Na slajdovima 21 i 22 vidimo kako se u Maven-u postavlja stroga granica. Konfigurišemo `jacoco-maven-plugin` i postavljamo pravilo `COVEREDRATIO` sa minimumom od `0.80`, odnosno 80 procenata.

[Slajd 23]

Šta se dešava kada programer napiše novu složenu metodu, zaboravi da napiše test i pokrene `mvn clean verify`? Pokrivenost pada na 78%, JaCoCo kapija pada sa porukom `BUILD FAILURE`, i pajplajn automatski blokira spajanje tog koda u glavnu granu.

[Slajd 24]

Kako projekat raste, javlja se novi problem: imamo hiljade testova i njihovo izvršavanje počinje da traje satima. Tada nastupa takozvani **zamor testova** — programeri gube strpljenje i počinju da preskaču provere. Zato nam treba optimizacija i selekcija testova.

[Slajd 25]

Jedan napredan način za to je **Change Impact Analysis**. Sistem analizira koje su tačno linije koda izmenjene u poslednjem Git commit-u, preko grafa kontrole toka pronađe koje klase zavise od tih izmena, i pokrene samo onaj podskup testova koji prolazi kroz taj deo koda.

[Slajd 26]

U praksi sa Maven-om ovo najčešće rešavamo razdvajanjem brzih od sporih testova. Brze jedinične testove pokrećemo na svaki commit komandom `mvn test`, dok spore integracione testove odvajamo preko `failsafe` plugina i pokrećemo ih tek pre spajanja koda komandom `mvn verify`.

[Slajd 27]

Na ovom slajdu imate kratak podsetnik ključnih pojmova koje smo do sada prošli: Build Pipeline kao automatizovanu traku, Pre-merge validaciju unutar Pull Request-a i SAST automatizovanu statičku proveru bezbednosti koda.

DORA metrike i zaključak (Slajdovi 28–37)

[Slajd 28]

Sada dolazimo do pitanja koje zanima svaki menadžment i svaki tim: ***kako mi zapravo merimo da li naše Kontinualno Testiranje daje rezultate?*** Odgovor leži u četiri standardne **DORA metrike**: učestalost isporuke, vreme realizacije izmena, stopa neuspešnih izmena i vreme oporavka sistema.

[Slajd 29]

DORA je skraćenica za ***DevOps Research and Assessment***, tim koji je godinama istraživao šta razlikuje najbolje svetske inženjerske timove od prosečnih. Ove četiri metrike se dele na dve strane medalje: prve dve mere **brzinu**, a druge dve mere **stabilnost**. Pravi DevOps cilj je povećati brzinu, ali bez žrtvovanja stabilnosti.

[Slajd 30]

Prva metrika je **Deployment Frequency**, odnosno koliko često puštamo kod u produkciju. Elitni timovi to rade više puta dnevno. Bez automatizovanih testova, timovi se plaše

produkcije i gomilaju kod u ogromne, rizične verzije. Brzi JUnit i Javalin testovi daju timu samopouzdanje da isporučuje male izmene svakog dana.

[Slajd 31]

Druga metrika je **Lead Time for Changes** — vreme od trenutka kada programer uradi commit do trenutka kada taj kod radi u produkciji. Glavni kočničar ovde je uvek bilo ručno testiranje koje traje danima. Pomeranjem testiranja ulevo kroz automatizovane kapije kvaliteta, ovo vreme spuštamo sa dana na svega nekoliko minuta.

[Slajd 32]

Treća metrika je **Change Failure Rate**, odnosno procenat isporuka koje izazovu pad ili zahtevaju hitnu popravku u produkciji. Upravo ovde statička analiza poput SpotBugs-a i JaCoCo kapije kvaliteta direktno obaraju procenat grešaka kod elitnih timova na ispod 5%.

[Slajd 33]

Četvrta metrika je **Time to Restore Service** — koliko brzo vraćamo sistem u život kada se incident ipak desi. Tu nam u pomoć priskaču Shift-Right tehnike: telemetrija preko Micrometer-a nas odmah alarmira čim skoči broj grešaka, a Feature Flags nam omogućavaju da problematičnu opciju ugasimo jednim klikom za nekoliko sekundi.

[Slajd 34]

Godinama je postojao večiti sukob: razvojni tim želi brzinu, a operativni tim želi stabilnost. Continuous Testing je most koji briše taj konflikt — automatizovane kapije nam dozvoljavaju maksimalnu brzinu, dok istovremeno matematički garantuju stabilnost koda.

[Slajd 35]

Ova tabela jasno pokazuje transformaciju: sa uvođenjem Continuous Testing-a prelazimo iz straha od greške i dugog čekanja u brze, česte i stabilne isporuke.

[Slajd 36]

Pogledajmo kako izgleda kompletan tok od početka do kraja: u prvoj fazi programer lokalno pokreće JUnit i SpotBugs; nakon Git push-a pokreće se CI server koji vrti statičku analizu i testove; zatim JaCoCo proverava pokrivenost kao kapija kvaliteta, i na kraju kod odlazi na produkciju pod kontrolom Feature Flag-ova.

[Slajd 37]

Da zaključimo današnje predavanje. Continuous Testing danas nije luksuz, već neophodnost. Kroz Shift-Left smanjujemo cenu bagova, kroz Shift-Right bezbedno pratimo sistem u produkciji, a kapije kvaliteta čuvaju integritet aplikacije bez usporavanja tima. Zapamtite: ***kvalitet se ne dodaje na kraju — on se gradi i proverava od prve linije koda do produkcije***. Hvala na pažnji!