Plugin pro sledování změn souborových výstupů pro systém průběžné integrace Jenkins

Michal Hrabovský

Bakalářská práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

*** Nascanované zadání, strana 1 ***

*** Nascanované zadání, strana 2 ***

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářské práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne .....

podpis studenta

ABSTRAKT

Práce se zabývá problematikou psaní pluginu pro automatizační systém Jenkins. V Teoretické části bude rozebrána funkce a účel systému Jenkins, možnosti které tento systém nabízí pro rozšiřování své funkcionality rozšířeními, a technologie které jsou pro tvorbu těchto pluginů využívány, jako například templatový system Jelly. Výsledkem praktické části je poté rozšíření systému Jenkins, který detekuje a prezentuje změny v souborových výstupech jednotlivých opakovaných spuštění Jenkinsové úlohy. Plugin je primárně určený k detekci neočekávaných nebo nežádaných změn ve výstupech úlohy za účelem regresního testování. Tyto změny jsou prezentovány v GNU unified diff formátu přímo v webovém rozhraní serveru systému Jenkins.

Klíčová slova: Jenkins, Java, Plugin, Systém průběžné integrace

ABSTRACT

This thesis concerns itself with the topic of creating a plugin for continous integration system jenkins. In the theoretical portion, it describes functionalities and purpose of the Jenkins system, the ways this system enables extending its capabilities with additional plugins, and the technologies that are used for creation of these plugins, such as the templating system Jelly. The output of the practical portion is a plugin for system Jenkins, which detects and presents changes in files outputted by individual runs of a Jenkins task. The plugin is primarily intended to detect unexpected or undesired changes in outputs of repeatedly ran tasks for purposes of regression testing. These changes are presented in GNU unified diff format directly within the Jenkins server's graphical interface.

Keywords: Jenkins, Java, Plugin, Continuous integration system

OBSAH

ÚVOD

První řádek prvního odstavce v kapitole či podkapitole se neodsazuje, ostatní ano. Vertikální odsazení mezy odstavci je typycké pro anglickou sazbu; czech babel toto respektuje, netřeba do textu přidávat jakékoliv explicitní formátování, viz ukázka sazby tohoto textu s následujícím odstavcem).

Formátování druhého odstavce. Text text text text text text text text text text text text.

I. TEORETICKÁ ČÁST

1 SYSTÉMY POSTUPNÉ INTEGRACE

1.1 Představení problému

1.1.1 Úvod

Týmy programátorů pracující na stejném projektu se potýkají s mnoha problémy, které vyvstávají z potřeby integrace výsledků jejich práce do sjednocené verze produktu, která je následně předávaná zákazníkům.

Tyto problémy se vyskytují v každé fázi procesu integrace od samotného slučování práce jednotlivých vývojářů přes testování izolovaných přídavků až po samotné nasazení předání výsledného produktu klientům.

1.1.2 Správa verzí

V jakémkoliv projektu, na kterém pracuje více vývojářů, vyvstává problém toho, jak sloučit jejich práci do výsledného produktu. Je nutné sledovat jaké změny byly provedeny v každé verzi programu jednotlivými vývojáři, a poté je zavést do jednotné společné verze. Jakákoliv chyba v této společné verzi je poté propagována k ostatním vývojářům, kde poté komplikuje jejich práci, nebo dokonce předána ve finálním produktu zákazníkům.

Tyto odlišné verze systému které vývojáři tvoří svými změnami se nazývají větve, společná hlavní větev se potom nazývá kmen, nebo tradičně 'master' větev. (Tento výraz bude použit později v práci)

Systémy správy verzí tuto činnost do velké míry automatizují, čímž sice nezamezí chybám v kódu jednotlivých přispěvatelů, ale pomáhá zamezit chybám při samotném slučování (integraci) [**?**]HandsOn)

Při slučování změn v produktu nastávají konflikty, pokud více vývojářů pracovalo na stejných částech kódu. Tyto konflikty musí být pracně manuálně řešeny slučujícím vývojářem, což vyžaduje pečlivou analýzu obou sad změn v kódu, a většinou obeznámení se s velkou částí změn provedených druhým vývojářem. Chyba v tomto procesu znamená potenciální potřebu reverze změn, a zopakování celého slučování. Alternativně se lze pokusit o opravení této chyby ve sloučené verzi, ale to dál komplikuje nalezení příčin špatného fungování kódu. Pokud bude chyba objevena později během vývoje, nebude očividné jestli je problém v změnách jednoho z developerů, nebo vznikl v samotné integraci.

Software správy verzí současně poskytuje distribuovaný sytém zálohování dat. Každý developer u sebe mívá většinou kromě své vlastní větve alespoň hlavní větev, její kopie se tak nacházejí na mnoha zařízeních. Ztráta hlavní větve v důsledku selhání systému správy verzí je tak velmi nepravděpodobná.

Ve větších projektech je navíc standardní praxí kontrola integrovaných změn druhou osobou. V tomto případě developer neslučuje svoji verzi programu sám, ale žádá a potvrzení od osoby pověřené kontrolou, která následně požadavek přijímá nebo zamítá.

1.1.3 Build/kompilace/linkování

V závislosti na komplexnosti projektu může kompilace zabírat velké množství času a výpočetních prostředků. Kromě celistvé verze programu může být třeba připravit instalační balíčky, vytvořit verze pro různé operační systémy, speciální verze programu určené pro různé typy testování, umístit všechny výsledky na servery (ať už jde o verze projektu pro veřejnost nebo interní účely), vygenerovat dokumentaci, upozornit vývojáře na existenci nové verze, a další administrativní úkony.

Manuální vykonání tohoto dlouhého seznamu úkonů může trvat den práce nebo víc i člověku který je s procesem už seznámen a ví přesně jak vše vykonat. Jedná se navíc o repetitivní druh práce, kterou vývojáři většinou nepříliš rádi vykonávají, což zve k nepozornosti a chybám způsobených jednotvárností činnosti.

1.1.4 Testování komponent

Při testování větších kusů programu může být obtížné přesně určit, která část je chybná. Může trvat celkem dlouhou dobu než rozpracovaná část programu dokáže interagovat se zbytkem projektu, a ve chvíli kdy tento stav nastane by mohlo být potřeba otestovat i několik stovek řádků nového kódu najednou. Je tedy problematické nemít po celou tuto dobu vývoje žádnou kontrolu výsledků.

Právě k tomu slouží Testování komponent, které izolovaně testují malé kusy produktu.

Pokud se jedná o nejmenší testovatelné části programu, jako například funkce nebo třída, často se takovým testům říká "jednotkové testy". Tyto rychlé testy často navrhují samotní vývojáři a ne testovací tým. Jsou prováděny během samotné práce, a okamžitě nahlašují špatnou funkcionalitu právě napsaného kódu. V paradigmatu vývoje řízeného testy jsou tyto testy dokonce psané před samotnými funkcemi které testují, a slouží zároveň jako definice toho jaké výstupy a vstupy by funkce měla mít.

Tyto testy jsou používány před procesem integrace, ne během něj. Všechny jednotkové testy by měly úspěšně proběhnout již před pokusem o integraci daného bloku práce, ale můžou být pro jistotu opakovány i během procesu integraci samotné. [1]

Testování komponent se ale také zabývá testováním větších samostatně testovatelných bloků programu. Takové testy se často koncipují jako black-box testy. Tz. neberou v úvahu implementaci testované komponenty, ale pouze kontrolují správnost výstupu při daném vstupu.

Toto testování většinou zaštiťuje testovací tým, ne tým vývojářů, a je prováděno na části systému, která je již zprovozněná a je schopná vykonávat svou funkci, přičemž se kontroluje jestli je daná komponenta schopná úspěšně obsloužit daný příklad jejího použití.

Komponentou může být například jedna webová stránka, jeden formulář na webové stránce, jedno okno či jedno menu desktopové aplikace, popřípadě jedna funkce programu zpouštěného z příkazové řádky.

Testování komponet lze rozdělit na dva podtypy.

CTIL - "Component testing in large" spočívá v testovaní komponenty v rámci zbytku systému. Stále se soustředíme na komponentu samotnou. Její správnou integrací se zbytkem systému se zabývá integrační testování. Mezi hlavní nevýhody patří náchylnost na falešná positiva v testech způsobených jinými komponentami, a také to, že lze metodu použít až po zprovoznění celého systému.

CTIS - "Component testing in small" spočívá v izolování komponenty od externích vlivů. Pokud má komponenta závislosti na jiných částech systému, je nutné nahradit potřebné komponenty náhražkami. Tyto náhražky se nazývají "stubs" pokud jsou používány testovanou komponentou, a "drivers" pokud testovanou komponentu využívají. Nevýhodou je náročnost přípravy testů pro tuto metodu.[2]

1.1.5 Integrační testy

Po tom co jednotkové testy ověří validitu jednotlivých drobných částí systému je třeba kontrolovat jejich vzájemnou komunikaci a spolupráci. K tomu slouží integrační testy, které testují větší části pogramu. [1]

Procesy integračního testování se od sebe navzájem liší v závislosti na zvoleném přístupu.

- Testování zvrchu-dolů (top-down) testuje prvně komponenty na špičce hiearchie a postupuje k podřazeným, takže rychle objeví problém a poté ho postupně izoluje.
- Testování zdola nahoru (bottom-up) testuje podřazené komponenty jako první a postupuje po hiearchii nahoru, takže může být prováděno bez toho aby byly nadřazené komponenty dokončeny.
- Ad-hoc testovaní spočívá v testování komponent v tom pořadí ve kterém jsou dokončovány, což může být méně pracné, ale může docházet ke ztrátě kontroly nad procesem testování a organizačním problémům.

Integrační testy můžou být prováděny black-box i white-box způsobem.

1.1.6 Systémové testování

Jak název napovídá, jde o testování celého systému jako celku. Proto bývá poslední fází testování.

Cílem je otestovat jestli celý integrovaný systém vyhovuje specifikovaným požadavkům. Typicky se zde může testovat rychlost systému, škálovatelnost, zátěžové testy, komunikace mezi více běžícími kopiemi softwaru, ale i jazykové lokalizace. [3]

1.1.7 Souhrn

Výše zmínená posloupnost jednotlivých úkonů představuje dosti velké úsilí vynalozené do ověření funkcionality každé verze programu byť se od predchozí lisí jen minimalně. To by byl problém sám o sobě čistě kvůli množství práce kterou to představuje, ale jedna položka konkrétně představuje další dilema.

Správa verzí softwaru se stává mnohem složitější podle toho, jak moc se od sebe liší verze, na kterých developeři pracují. Práce každého člena vývojového týmu musí být integrována do verze projektu, která se výrazně liší od verze, ze které vývojář vycházel. To znamená, že buďto během jeho práce musí tým buď zmrazit část projektu, na které se pracuje, aby se závislosti nového příspěvku neměnily mezitím co práce probíhá, nebo musí tým komunikovat natolik, aby všechny změny byly kompatibilní.

Pokud se nepodaří zajistit kompatibilitu změn developera a ostatních změn v centrální sdílené verzi programu, musí být program měněn v rámci samotného procesu slučování. To pravděpodobně znamená přenesení změn z centrální větvě programu do větve developera, opětovné otestování této výsledné verze a opakovaný pokus o sloučení.

Pokud chceme omezit množství problémů s původem ve správě verzí, musíme tedy mít malé množství změn v každé verzi, a tyto verze vytvářet často. S každou verzí je ale nutné provést celý integrační proces, jehož součástí je testování a vydání verze, takže celkové množství práce neklesá. Správa verzí a zbytek úkonů jsou tímto způsobem navzájem v opozici.

Odpovědí na toto dilema byl vývoj systémů a praktik jejichž zodpovědnost je tento integrační proces pro vývojáře zjednodušit a zrychlit, a zrodil se tak koncept průběžné integrace.

1.2 Principy průběžné integrace

1.2.1 Cíle

Samotný název průběžná integrace(anglicky **continuous integration** , zkratkou **CI**), napovídá co je hlavní cíl. Umožnit, aby developeři mohli provádět integrační procesy

soustavně **během** práce, oproti tomu aby to byla aktivita která sama o sobě tvoří celou koncovou etapu vývoje dané komponenty. Tyto časté integrace výrazně snižují náročnost těch částí integračního procesu, které se bez dedikované pozornosti vývojáře vykonat nedají.

Koncept Continuous Integration tohoto záměru dosahuje extensivní automatizací. Časté opakování těchto úkolů není problémem pokud probíhají automatizovaně bez zásahů developerů. Toto je aplikováno do té míry že CI systémy často spolupracují se systémy správy verzí a nabízí možnost automaticky odstartovat dané procesy, když je v nich vytvořena nová verze software.

U větších projektů může být celý proces náročný i na strojový čas, takže další běžnou funkcionalitou bývá možnost rozdělování úkolů více strojům které je poté vykonávají současně. Toto urychluje zpětnou vazbu ohledně stavu verzí, a může být přímo nutné, protože je velmi možné že jeden stroj by ani nestíhal zpracovat veškeré požadavky na integraci od všech developerů. Rychlost je obvzlášť důležitá v oblasti testování, kde rychlá zpětná vazba výrazně urychluje proces opravy chyb.

Tyto systémy, které se často skládají z mnoha dedikovaných strojů, služeb a třeba i licencovaných vyžadují mnohdy nezanedbatelné množství finančních a organizačních prostředků, ale tyto náklady se odráží v redukcích času potřebného pro vývoj softwaru.

Continuous integration zároveň řeší některé další problémy spojené s různými kroky integrace.

1.2.2 Správa verzí

Jak už jsem rozebral, častá integrace samotná inherentně snižuje potřebné množství zásahů ze strany developerů během slučování verzí softwaru. Kromě toho průběžná integrace pomáhá rychleji odhalovat chyby a zmenšuje tak rámec kódu ve kterém se chyba pravděpodobně nachází. Continous integration systémy často nabízejí integraci s různými běžnými version control systémy. Ta umožňuje nastavit tzv. "hook", automatickou komunikaci mezi CI systémem a systémem správy verzí. Na základě této komunikace CI systém poté automaticky provádí integrační procesy s novými verzemi softwaru hned poté, co jsou do version control systému přidány.

Co se týče samotné integrace, jedno z častých použití continuous integration systému je zajištění bezchybnosti hlavní větve softwaru. Developeři si ustanovují pravidlo že jakýkoliv příspěvek do hlavní větve musí projít testováním před jeho schválením. Toto testování je zahájeno integračním systémem v okamžiku kdy je vytvořen požadavek o přidání změn z větve developera do hlavní větve. Většinou je toto testování intenzivnější než testování během vývoje samotného, vývojový proces například může používat pouze jednotkové testy během vývoje, a integrační testy nebo component tes-

ting in large provádět pouze když se změny integrují do hlavní větve. To má samozřejmě smysl pouze tehdy pokud je kompletní sada testů příliš výkonově náročná na to, aby se prováděla s menšími změnami které developer provádí ve své větvi.

1.2.3 Build

Continuous integration systémy poskytují unifikované prostředí pro proces sestavení programu. Díky tomu zabraňují případům, kdy lze verze programu sestavit na straně developera, ale odlišnostem v prostředí není sestavitelná univerzálně. Příčinou je většinou to že developer během vývoje vykonal nějaké změny na svém stroji, aby program sestavil, ale tyto změny nereprezentoval příslušně v různých skriptech a souborech, které doprovázejí zdrojový kód programu.

Toto nemusí řeší problém předchozích zásahů do prostředí stroje na kterém sestavení běží v rámci continuous information systému. Tento problém lze dále potlačit jinými systémy. Příklad populárního řešení je systém Docker, který umožňuje sestavení provádět v explicitné definovaném virtuálním prostředí, které bude vždy stejné nehledě na prostředí systému, na kterém docker běží.

Dalším faktorem je výpočetní výkon, který lze v continuous integration systému efektivněji využívat díky tomu, že je hardware sdílený mezi několika developery. Díky nezávislosti na prostředí lze poté hardwarové prostředky rozšiřovat na základě požadavků projektu, nebo je sdílet i mezi několika projekty/týmy. Přístup k výkonným zařízením pak může snižovat čas sestavení u náročných projektů a zrychlovat celý proces vývoje, a to za nižší náklady než dodávání strojů jednotlivým developerům.

1.2.4 Testy

Prakticky všechny body které se týkají sestavení programu se týkají i testování. Obzvláště co se týká výkonu, protože doba mezi opravou chyby a ověření výsledků je zásadní pro rychlost vývoje a udržování kvality softwaru.

Automatizované řešení mají za následek stabilnější pokrytí testů, protože se nemůže stát že část z nich bude opomenuta kvůli lidskému faktoru. Doporučované je i používání různých nástrojů na zjištění pokrytí které pomáhají udržovat kompletnější sadu testů. [4]

1.2.5 Deploy

V oblasti deploye pomáhá continuous integration se šetřením pracovní síly. Systém by se měl starat o obnovení dokumentace a poskytnutí sestaveného programu vývojářům. Dalšími úkony jsou například sestavení testovacích reportů nebo rozeslání notifkací/emailů všem zainteresovaným zaměstnancům.

Prostředí kdy jsou tyto úlohy zvlášť důležité jsou ale aktualizace již běžících systémů. V těchto případech je rychlý a hladký proces deploye zásadní, a u systémů které musí být spuštěny nepřetržitě (stylu internetové služby které fungují na více serverech), se může komplikovat ještě například potřebou přesměrovávat požadavky od částí systému která je právě aktualizována.

Druhý typ problematických systémů můžou být systémy kde je nutná velká míra spolehlivosti, jako například systémy internetového bankovnictví. Zde není přípustné aby chyba vyvstávající ze samotného procesu deploye způsobila neočekávané chování systému, a systém musí být otestován přesně v tom stavu v jakém bude přístupný uživatelům. To znamená přerušení činnosti systému, které zároveň musí být co nejkratší.

Jedna z rolí Continuous integration systému v oblasti deploye by tedy měla být zkrácení nebo eliminování doby nedostupnosti služeb během jejich aktualizací. Druhou rolí je zajištění toho že developeři pracují vždy s aktuálními informacemi a čerstvou dokumentací.

2 PŘÍKLADY CI SYSTÉMŮ

2.1 Travis CI

Travis CI je placený proprietární continuous integration systém který upřednostňuje jednoduchost prvotního nastavení systému, a přístup k CI systému bez nutnosti zavádět a udržovat infrastrukturu.

Jedná se o primárně cloudové řešení, které provádí žádané operace na automaticky přidělovaných virtuálních strojích.

Travis je použitelný pro Windows, Linuxm, MacOS a FreeBSD projekty, a podporuje v základu integraci s VCS systémy GitHub, Bitbucket, Assembla, a GitLab. [5]

2.1.1 Concurrency based plan

Travis nabízí dva způsoby použití a placení. Prvním je "Concurrency based plan", který umožňuje provádět stanovený počet paralelních buildů najednou. Buildy které tento limit přesáhnou musí na své provedení čekat ve frontě dokud se již běžící buildy nedokončí. Jedná se tak víceméně o rezervaci přístupu k určitému počtu strojů v cloudu, a tyto stroje můžou být neomezeně využívány bez dopadu na cenu předplatného.

2.1.2 Usage based plan

Druhým módem je "Usage based Plan"který umožňuje přístup k neomezenému množství strojů, ale smlouva zahrnuje omezený počet 'kreditů'. Spouštění buildů stojí určitý počet kreditů na základě toho kolik strojového času build zabere, a na jakých zařízeních se provádí. (Druh zařízení určuje uživatel sám v konfiguračním souboru Travisu) Dále je třeba měsíčně platit kredity za každého uživatele který buildy spouští. Tento plán tak umožňuje build provést v nejrychlejším možném čase, zvlášť pokud se build skládá z více paralelizovatelných částí. Rozklad na paralelní části lze provést intenzivně např. v testování, kde na sobě testy bývají často navzájem nezávislé. Kredity je možné kdykoliv dokupovat, nebo je získávat v rámci pravidelného předplatného. Cena provozu Travis CI v rámci tohoto plánu se tedy odvozuje od míry využití.

2.1.3 Travis Enterprise

Travis lze používat i na vlastní infrastruktuře a vlastním hardwaru, a to ve formě 'Travis Enterprise'. Toto vyžaduje měsíčně platit za licenci pro každého uživatele. Travis Enterprise také podporuje pouze GitHub a GitHub enterprise, ne ostatní Version Control systémy. /citetravis-enterprise-doc

2.1.4 Free

Kromě placených režimů nabízí travis CI i free plan který ale zahrnuje pouze jednorázovou sumu kreditů pro každého nového uživatele, takže je vhodný spíš k vyzkoušení platformy než k jakémukoliv dlouhodobému využívání.

2.1.5 Výhody a nevýhody

Tyto charakteristiky bohužel mají za následek určitá omezení co se týče přizpůsobivosti systému Travis CI. Virtuální stroje v cloudu musí mít jednu z omezeného množství počátečních konfigurací.

Dalším problémem může být bezpečnost, protože mnohé akce buildu vyžadují aby různá hesla a tajné tokeny opustily prostředí sítě společnosti. Zejména free plan umožňuje komukoliv přístup k velkému množství senzitivních dat uživatelů. [6]

Travis Enterprise tento problém nemá.

2.1.6 Souhrn

Pohodlnost použití, nulové požadavky na zázemí a možnost "platit jen co se použije"znamenají že Travis a podobné cloudové CI systémy jsou atraktivní pro menší společnosti nebo hobby projekty pro které by zavedení a udržování podobně rychlé infrastruktury znamenalo neúměrné množství nákladů.

Z toho důvodu byl Travis byl velmi populární CI systém v opensource projektech a na platformě GitHub, [7] ale od roku 2018 mu na této platformě silně konkuruje CI systém 'GitHub Actions' vyvíjený GitHubem samotným.

2.2 CircleCI

CircleCI je nápodobně placený proprietární continuous integration systém.

CircleCI podporuje použití operace v cloudu i na místních strojích, a pro účely debugování částečně i na osamoceném počítači mimo systém CI. Exekuce může běžet na Windows, Linuxu, MacOS a Androidu. [8]

2.2.1 Platební plány obecně

Podobně jako Travis Circle CI zavádí Circle CI kreditový systém, kde jsou kredity utráceny za minuty strojového času strávené buildem.

Vyjimkou je Circle CI server, který podobně jako Travis CI enterprise umisťuje celou infrastrukturu do prostředí spravovaného zákazníkem.

Stejně jako Travis nabízí výběr výkonu stroje v cloudu, a stejně tak platí že výkonnější stroje stojí větší množství kreditů za minutu. Navíc je možné že výkonnější konfigurace nebudou okamžitě k dispozici, a že bude uživatel muset čekat ve frontě. Při nižším vytížení může být naopak build přiřazen výkonnějšímu stroji v cloudu, takže se jedná spíš o určení 'minimálního' požadavku.

2.2.2 Běh na vlastním stroji

Na rozdíl od Travis CI umožňuje Circle CI použití lokálních strojů v rámci všech svých platební plánů. Počet těchto strojů je omezený v závislosti na tom jaký platební plán je používán.

Toto neznamená že je celá infrastruktura pod kontrolou uživatele. Hlavní server CI je v tomto módu používání stále spravovaný Circle CI samotným. Lokální stroje se podřizují tomuto externímu serveru, a spouštějí procesy podle jeho pokynů. Server běžící na straně zákazníka lze mít pouze skrze Circle CI server business plán.

Tímto CircleCI pokrývá některé situace kdy má uživatel specifické nároky na prostředí ve kterém buildy či jiné operace probíhají.

Běh operací na vlastním stroji nestojí žádné kredity, ale platí se kreditový poplatek za přenos dat mezi Circle CI serverem a vlastním strojem a za úložný prostor využívaný na Circle CI serveru (ten se používá například na uchovávání některých dat nebo celého prostředí mezi několika spuštěními úlohy bez toho aby úloha musela být vykonána vždy na stejném stroji).

Ne všechny typy systémů také mají stejnou úroveň podpory od Circle CI. V dokumentaci se uvádí že self-hosting je testován na macOS X 11.2+ (Intel, Apple M1), Windows Server 2019, 2016 (64bit), a Linux distribuc9ch - RHEL8, SUSE, Debian, etc (x86_64, ARM64, s390x, ppc64le). Ostatní typy systémů nejsou plně podporovány a CircleCI negarantuje asistenci s problémy spojených s jejich používáním.

2.2.3 Free plán

Circle AI nabízí zdarma 6000 strojových minut měsíčně na nízkovýkonovém virtuálním docker stroji (výrazně méně se silnějšími zařízeními). Narozdíl od placených plánů nemá tento plán omezení na počet uživatelů, ale kreditový strop stejně omezuje velkoplošné využití.

Free plán je omezen na 30 paralelních jobů, a 5 vlastních strojů.

S Free plánem také nelze využívat některé typy zařízení (např výkonnější třídy MacOS strojů)

2.2.4 Placené plány

Circle CI má dva normální placené plány. Performance a Scale.

Performance začíná na 15 dolarech za měsíc a zahrnuje dvojnásobné množství kreditů než free plan. Performance plán může používat 5 uživatelů, více uživatelů vyžaduje příplatek v kreditech, který vychází na 15 dolarů na uživatele.

Performance zahrnuje také vyšši prioritu v čekacích frontách, využití většího počtu paralelních strojů, přístup ke strojům skrze přidělené IP adresy, a využití k výkonově silnějším třídám strojů (ale stále ne ke všem).

Performance plán je omezený na 30 paralelních jobů, a 20 vlastních strojů

Scale plán zahrnuje všechny tyto funkcionality, ale navíc i přístup k nejvýkonnějším třídám strojů, a k GPU-centrickým strojům pro výpočetně složité úkony prováděných na grafických kartách.

Cena Scale plánu začíná na 2000 dolarech, a množství kreditů a uživatelů závisí na konkrétní smlouvě.

2.2.5 Circle CI server

Posledním plánem je Circle CI server který běží plně na infrastruktuře zákazníka. Cena a detaily tohoto plánu závisí na konkrétní smlouvě.

2.2.6 Orbs

"Orbs"jsou funkcionalita Circle CI která umožňuje zabalit části definice úkonů do parametrizovaných znovupoužitelných celků.

Toto usnadňuje nastavování některých běžně potřebných úkonů, stylu posílání notifikací skrze některé aplikace (Slack, Microsoft Teams apod.), nebo deploy na určité platformy (AWS).

Orby tak efektivně fungují jako lehké pluginy které rozšiřují funkcionalitu Circle CI nebo standardizují procesy v rámci organizace. Orby ale neumožňují dělat cokoliv co by nešlo běžným způsobem vypsat do configuračního souboru Circle CI.

Orby může tvořit kdokoliv, soukromé nebo veřejné, a jsou kategorizované do tří kategorii - Orby tvořené a ověřené Circle CI samotným, orby tvořené partnery Circle CI, a orby tvořené komunitou. Orby je tak možné rozlišit podle důvěryhodnosti.

Orby lze vyhledávat skrze oficiální registr orbů který Circle CI udržuje.

2.2.7 Souhrn

CircleCI nabízí atraktivní hybridní přístup mezi cloudovým a místním řešením. Orby s sebou přináší výhodu

2.3 Jenkins

Jenkins je velmi odlišný od předchozích dvou příkladů. Jedná se o open-source projekt psaný v jazyku Java, poprvé vydaný v roce 2011. Jako open-source projekt nenabízí žádnou možnost pronajímat infrastrukturu, a uživatel tak musí dodat vlastní hardware pro své CI účely. Na druhou stranu je Jenkins používání Jenkins naprosto zdarma.

2.3.1 Infrastruktura

Jenkins se zkládá ze serveru a jednoho nebo více exekutorů. Na serveru běží webové uživatelské rozhraní pomocí kterého lze systém nastavovat a ovládat. Server také funguje jako svůj vlastní exekutor (nebo několik exekutorů), takže systém lze v krajním případě provozovat na jednom počítači.

2.3.2 Pluginy

Jedna z hlavních výhod CI Jenkins je přistupnost mnoha (téměř 1900 v době psaní práce) pluginů které můžou rozšířit jeho funkcionality. Mezitím co Orby v Circle CI mohou dosáhnout pouze toho co je definovatelné v konfiguračních souborech, Jenkins pluginy jsou psané v Javě a napojují se přímo do infrastruktury Jenkinsu a mohou měnit GUI, zasahovat do celého systému, rozšiřovat integraci Jenkinsu s jinými systémy, a mnoho dalších funkcionalit. [9]

Jenkins bez žádných pluginů je **velmi** malý systém co se týče různých funkcí a dovedností. Minimální instalace jenkinsu prakticky umožňuje pouze pouštět skripty v příkazové řádce na Jenkins serveru (nedokáže kontrolovat jiné stroje) a disponuje čekací frontou na úlohy.

Designová filozofie Jenkinsu je taková že prakticky všechny funkcionality které má jsou dodávané pluginy, ať už od projektu Jenkins nebo od komunity. Uživatel si poté vybírá pouze ty funkcionality které chce využívat.

K těmto účelům umožňuje Jenkins tvořit pluginy pro pluginy, a celý systém tak funguje způsobem "Postavte si vlastní Continuous Integration". Jádro Jenkinsu záměrně obsahuje jen naprosté minimum k tomu aby s ním šlo vůbec něco vykonat. Některé funkcionality které zde uvádím tak nejsou součástí jádra Jenkinsu, ale jsou přesto k nalezení ve většině Jenkins instancí které jsou aktivně využívány.

2.3.3 Jenkins s Cloudem

Jenkins sice není vlastněný žádnou společností která by jednoduchým způsobem poskytovala přístup do svého cloudu, to ale neznamená že exekutoři nemohou být umístěny v pronajatém cloudu. Například plugin 'Amazon Elastic Container Service (ECS)' umožňuje dynamicky rezervovat virtuální stroje v Amazon Cloudu podle momentální zátěže systému jenkins. Plugin 'Azure VM Agents' zase zprostředkovává interakce se systémem Microsoft Azure. [9]

Další možností jsou cloudové systémy hostované samotným uživatelem, k těmto účelům další pluginy poskytují integraci s Kubernetes clustery, a vSphere virtualizovaným cloudem od společnosti VMware. [9]

V oblasti škálování je také významná společnost Cloudbees. Cloudbees je sponzor projektu Jenkins, a spravuje vlastní sadu pluginů která pomáhá spravovat a aplikovat Jenkins ve velkém měřítku. Cloudbees operuje nad více Jankins mastery, a umožňuje dynamické přidělování exekutorů těmto masterům v situacích kdy by jeden centrální Jenkins master s mnoha exekutory způsoboval organizační a technické problémy mezi různými projekty a týmy. S pomocí Cloudbees mohou mít jednotlivé týmy nebo projekty svůj vlastní Jenkins master nad kterým mají kontrolu, ale zároveň zároveň využívat výpočetní prostředky sdílené celou organizací.

2.3.4 Souhrn

Jenkins exceluje ve oblasti flexibility. Jeho zavedení může být relativně složitější než u předchozích CI systémů, protože nedisponuje některými funkcionalitami které předchozí CI systémy mají v základu. Tyto funkcionality jsou ale k dispozici skrze pluginy a integraci s dalšími systémy.

Tato flexibilita zůstává relevantní i po zavedení systému, kdy se Jenkins snadno může přizpůsobit novým požadavkům a nečekaným změnám v situaci uživatele.

3 JENKINS V DETAILU

Pro popis samotného pluginu je napřed potřeba popsat pojmy a konceptů které se v rámci systému Jenkins používají. Začnu od organizace úloh a budu pokračovat přes infrastrukturu k technologiím na kterých je Jenkins stavěn.

3.1 Organizace úloh

3.1.1 Job

'Job' reprezentuje opakovatelnou sekvenci úloh kterou Jenkins vykonává. Každý Job má unikátní jméno.

Job může být definován několika způsoby, ale hlavní metody jsou "Freestyle project"definovaný pomocí nastavení skrze GUI Jenkinsu, a "pipeline"definovaná konfiguračním souborem typu Jenkinsfile (o těch později). Další speciální typy můžou být přidávány pluginy.

Job může buď obsahovat veškeré nastavení potřebné pro jeho spuštění, nebo může být parametrizovaný, což umožňuje upravovat některé jeho aspekty s každým individuálním spuštěním.

3.1.2 Build

'Build' je jedno spuštění jobu. V základu jsou identifikovány svým pořadovým číslem, ale můžou jim být přidělovány i textová jména.

Build udržuje několik informací. Vždy obsahuje log který byl během spuštění jobu vypsán, ale kromě toho má i svůj status v klasické formě OK/WARNING/FAIL (Více stavů lze opět přidat pluginy). Pokud je job parametrizován, build bude uchovávat parametry se kterými byl spuštěn.

Build dále může uchovávat 'artefakty', soubory které jsou uloženy na Jenkins serveru a uschovávány dokud není build smazán.

Další druhy dat jsou přidávány pluginy, typickými případy jsou reporty skládané z dat které generují různé testovací frameworky, nebo informaci o verzích softwaru se kterými byl build prováděn (např. git commit).

Do logu/konzolovému výstupům a do některých typů informací lze nahlížet už za běhu buildu, jiné jsou dostupné až po jeho dokončení. Build může být kdykoliv manuálně přerušen.

3.1.3 BuildStep

BuildStepy jsou jednotlivé akce které Jenkins vykonává. Typickým zástupcem buildstepu je např. vykonání příkazu v příkazové řádce systému. Některé buildstepy jsou proveditelné pouze v určité části buildu, a mohou sloužit k mnoha účelům, od vykonávání akcí ve workspace, přes záznam informací o buildu, nebo kontaktování externích systémů. Stejně jako Build mají svůj status typu OK/WARNING/FAIL, přičemž build dědí nejhorší ze statusů buildstepů ze kterých se skládá.

3.1.4 Exekutor

'Exekutor' je vykonavatel buildů. Joby si určují na kterých exekutorech nebo skupinách můžou být prováděny, a při spuštění jobu jsou jejich části přidělené volným exekutorům z tohoto výběru nebo zařazeny do fronty.

Na jednom exekutoru může běžet právě jeden build (ale build může běžet na více exekutorech najednou pokud job obsahuje paralelní prvky). Exekutor také neznamená jeden stroj, jeden stroj může poskytovat systému několik exekutorů. to může pomoct využívat plný výkon stroje pokud joby mohou využívat pouze omezený počet jader, nebo pomoct lépe rozdělovat zátěž systému mezi stroje pokud stroje nemají stejný výkon.

3.1.5 Workspace

Exekutoři přidělují každému jobu jehož build je na nich spouštěn jednu složku na svém disku - jejich workspace. Tato složka je sdílená mezi jednotlivými buildy tohoto jobu které proběhnou na tomto exekutoru, ale není sdílená (v základu) mezi exekutory.

To znamená že pokud může job běžet na několika exekutorech, není zaručené že build bude běžet se stejným workspace jako build který mu předcházel.

Joby by tedy měly být konfigurovány tak aby na sobě jednotlivé buildy neměly závislosti, nebo aby tyto závislosti byly předávány jinými metodami než pouze skrze jejich workspace. Dobrá praktika je nastavovat joby tak aby neměli závislosti na stavu svého workspace obecně, a stahovali veškerá potřebná data podle potřeby.

Důležité je že obsah workspace zůstává na zařízení dokud není manuálně vyčištěn, takže každý job v systému zabírá n x počet-možných-exekutorů fyzického místa. Z tohoto důvodu může být žádoucí workspace na konci každého buildu vymazat, i když zanechávání určitých dat ve workspace může urychlit exekuci buildu.

Workspace tak obecně není vhodný pro archivaci výsledků buildu ve formě souborů, protože jeho obsah nezaniká s buildem a výsledky můžou být roztroušeny po několika strojích.

K tomuto účely slouží artefakty nebo integrace s externím systémem.

3.1.6 Artefakt

Soubory z workspace můžou být nahrány na Jenkins server a uchovávány společně s ostatními záznamy o buildu. Tyto 'artefakty' zanikají když je build smazán, ale dokud build existuje tak jsou přístupné pluginům, skrze GUI nebo skrze webové API Jenkinsu.

Artefakty mají svou vlastní složkovou strukturu, která typicky koresponduje 1:1 s jejich umístěním we workspace.

3.1.7 Pipeline

Přestože jednodušší joby můžou být definovány způsobem "Freestyle project", Složitější konfigurace je vhodné definovat způsobem "pipeline". Jsou pro to tři důvody.

Freestyle projekt je jednak převážně lineární, mezitím co pipeline umožňuje složitější konfigurace a paralelismus.

Druhý důvod je že Pipeline je definován pomocí Jenkinsfile souboru, a konfigurace je tak vázaná ke zbytku projektu, a ne pouze uložena na Jenkins serveru.

Další důvod je přehlednost. Během nastavování freestyle jobu jsou všechny buildstepy zobrazeny pod sebou, a neuspořádany do žádných logických celků. Pokud jsou buildstepy používáné v pipeline, jsou většinou dále organizované pod blokem 'stage' což je sekvence buildstepů které jsou si sémanticky blízko. Textová reprezentace nastavení všech stepů navíc bývá mnohem kompaktnější než nastavení jobu v GUI, které může s počtem stepů výrazně bobtnat.

Pipeliny samotné lze definovat dvěma způsoby, jako deklarativní pipeline a scripted pipeline.

Scripted pipeline využívá skriptovací jazyk groovy. Skript je prováděn sériově a úkony které má jenkins provádět jsou definovány tagem 'stage'. Tag 'parallel' umožňuje vzít několik Stage objektů uspořádaných v datové struktuře typu Map, a provést je parallelně.

Deklarativní pipeline používá syntax definovanou jenkinsem. Je uspořádáná do do sebe navzájem vnořených bloků. Tato definice je převážně statická, a neumožňuje tolik flexibility jako scripted pipeline, ale její výhodou je větší přehlednost a relativní jednoduchost na pochopení.

Původně byly v jenkinsu k dispozici pouze scripted pipeliny, deklarativní pipeline je novější přídavek. V dnešní době jsou preferovanější spíš deklarativní pipeliny kvůli přehlednosti a větší podobnosti s ostatními continuous integration systémy.

Mezitím co deklaratiní pipeline je vždy relativně intuitivně pochopitelná, složitější

příklady scripted pipeline k pochopení potenciálně vyžadují pokročilou znalost jazyka groovy. Znalost pipeline potřebují k práci různé typy developerů, takže restrikce deklarativní pipeline mohou být výhodou.

Většinou je tedy deklarativní pipeline první volbou, a skriptovaná pipeline je používána pouze pokud není možné žádanou funkcionalitu implementovat pomocí deklarativní pipeline.

Pipelines jsou technicky jeden z pluginů Jenkinsu, přestože je to officiální plugin projektu, ne jeden z mnoha vytvořených komunitou, takže v naprosto očesané instalaci je nenaleznete, ale v profesionálním prostředí se Jenkins používá bez pipelines pouze zřídka.

3.2 Zpřažené technologie

Jenkins je stavěn na několika dalších technologiích se kterými jsem musel pracovat abych sestavil tento plugin.

3.2.1 Maven

Maven je project management systém který je používán při kompilaci Jenkinsu a jeho pluginů.

Maven používá koncept Project Object Model, soubor ve formátu XML který popisuje jak sestavit daný projekt.

Jenkins nabízí template projekt pro maven který zjednodušuje založení nového pluginu, ale vše co je potřeba je pom.xml, který udává jméno pluginu, verzi jenkinsu pro kterou je plugin stavěný, a verzi Javy která se má pro něj použít, a udává svého rodiče, který by měl být některá verze POMu 'org.jenkins-ci.plugins'.

Tento rodičovský POM má v sobě všechny potřebné závislosti a informace potřebné po sestavení Jenkinsu, takže maven dokáže pouze s těmito informacemi sestavit celý projekt.

Samotná funkcionalita pluginu je potom definována .java soubory které musí být umístěny v pevně daném místě ve složce src (src/main/java/io/jenkins/plugins/<jméno pluginu>).

Druhou důležitou lokací je resources složka (src/main/resources/io/jenkins/plugins/<jméno pluginu>), kde jsou umístěny hlavně různé HTML teplates a ikony které plugin potřebuje pro svou vizuální reprezentaci v GUI.

Pro správnou funkci mavenu samotného by měla být nastavena environment proměnná JAVA_HOME s hodnotou která je cestou k používanému JDK. Způsob nastavení environment proměnných se liší systém od systému a měl by být snadno dohleda-

telný.

Posledním požadavkem je Maven samotný, který lze stáhnout z oficiálních stránek https://maven.apache.org/. Maven lze umístit kamkoliv na disk, i když je vhodné jeho bin složku přidat do proměnné PATH.

Po nastavení těchto nutných komponent stačí vykonat v shellu příkaz mvn hpi:run s working directory ve složce projektu. Pokud jsou všechny nastavení a projekt samotný v pořádku, maven spustí Jenkins server na portu 8080, přístupný skrze adresu 127.0.0.1:8080/jenkins.

3.2.2 Jelly

Jelly je template engine který v Jenkinsu zprostředkovává definici a renderování GUI prvků. Jakékoliv GUI prvky které jsou pro plugin tvořeny musí být napsané právě pomocí Jelly.

Jenkins definuje velké množství svých HTML tagů pro jelly. Tyto tagy jsou popsány v dokumentaci Jankinsu (momentálně https://reports.jenkins.io/core-taglib/jelly-taglib-ref.html), a musí být používány aby plugin zapadaly graficky do zbytku Jenkinsu, a používání normálních HTML tagů ohrožuje správnou funkcionalitu pluginu.

3.2.3 Jenkins Extension pointy

Samotná funkcionalitu pluginu je třeba definovat skrze Extension pointy. Jsou to Java třídy které jsou používány na určitých místech v jenkinsu, a mají určitá rozhraní které developer pluginu musí implementovat aby se části pluginu správně objevily a fungovaly uvnitř Jenkins GUI.

Tyto Extension pointy musí být dekorovány dekorátorem @Extension jádro Jenkinsu takto dekorované třídy v plginu vyhledává a používá je na určitých místech. Extension point dále může použít argumenty které obdržuje na svém rozhraní aby dále ovlivnil chování Jenkinsu.

Například třída Recorder slouží k implementaci nového teststepu který zaznamenává data o buildu.

Tyto Extension pointy jsou popsány v dokumentaci Jenkinsu včetně extension pointů vybraných pluginů.

II. PRAKTICKÁ ČÁST

4 NÁVRH

4.1 Motivace

Púvodní impuls pro tvorbu pluginu byla nutnost odhalit drobné nečekané změny během regresního testování velkého množství C kódu kódu generovaného programem Config Tools od firmy NXP.

Během regresního testování byly generovány tisíce souborů v několikaúrovňové adresářové struktuře, takže manuální review nepřicházelo v úvahu.

Případná změna musela být viditelná už na úrovni Jenkins jobu který tyto testy prováděl, protože podobných jobů bylo spouštěno během testovacího cyklu několik desítek. Kód byl kontrolován osobně když byly tyto funkcionality prvně přidávány, ale regrese byla kvůli škále problém.

4.2 Inspirace pluginem Artifact-Diff

Pro Jenkins již existoval plugin 'Artifact-diff' od Olivera Gondži. Tento plugin přidává možnost na požádání vygenerovat diff mezi dvěma artefakty různých buildů pomocí zadání URL adresy do prohlížeče.

Projekt původně začal jako modifikace tohoto pluginu, a pořád nabízí tuto funkci, ale pro potřeby regresního testování byl tento plugin nedostačující. Plugin byl výrazně rozšířen o mnoho dalších funkcí, a i samotný proces tvorby diffů byl od základu přepsán kvůli potřebě identifikovat počet sekcí v kódu které byly změněny a filtrem ignorovat některé řádky kde byly změny očekávány.

Ze strany front-endu byl jeden jelly template napsán tak aby zhruba imitoval vzhled tohoto pluginu. Kód je ale originální, protože jsem nedokázal porozumět implementaci původního templatu.

Rozdíly jsou tak zásadní že se mi nepodařilo najít jediné místo kde by kód vypadal stejně. Ale považuji stále za vhodné zmínit že tento plugin byl inspirací a že se v mém projektu mohou nacházet drobné útržky stejného nebo alespoň podobného kódu.

4.3 Požadavky

Z účelu pluginu jsem identifikoval několik požadavků.

- Plugin musí svoji činnost vykonávat v průběhu buildu, ne na požádání po jeho dokončení.
- Plugin musí označovat buildy varovným nebo horším označením v případě detekce změny.

- Plugin musí poskytovat report na stránce jobu nebo buildu o množství detekovaných změn, pro případ že build skončí ve stavu Warning z jiného důvodu.
- Kvůli hloubce souborového systému je potřeba aby plugin reportoval v jakých částech testu byly změny nalezeny, tz. nelze vypisovat pouze seznam změněných souborů. Jedna chyba může vyvolat změnu v mnoha souborech, takže výpis všech změněných souborů neposkytuje dobrý přehled o chybách ktereé nastaly. Záměr je aby bylo možné odlišovat jeden typ chyby od druhého pomocí umístění v souborovém systému.
- Není nutné ani žádoucí srovnávat všechny soubory které build ukládá jako artefakty. Plugin potřebuje možnost filtrovat soubory podle cesty, názvu a přípony.
- Soubory mají některé části které se mění build od buildu, primárně časové údaje. Tyto části je nutné odfiltrovat.
- V případě kontroly výsledků několika buildů najednou nestačí porovnání pouze s posledním buildem. Typický je případ kdy se vyskytne nějaká chyba, je opravena, a výsledek po opravě je potřebné srovna s výsledkem před výskytem chyby. Je tedy potřebné na požádání srovnat dva libovolné buildy.

Většina požadavků na systém tedy nespočívá v porovnávání souborů, ale v organizaci, přehlednosti a konkrétních use cases.

4.4 Use cases

Nápodobně jsem zohlednil některé use cases, abych si ustanovil priority co se týče designu pluginu.

- Po updatu testovaných dat se objevilo 200 chybných testů roztroušených po 4 jobech. většina z nich je působena jednou chybou která je identifikována a nahlášena developement týmu. Chyba zůstane ale v testech alespoň po dobu týdne do dalšího deploye. Chybová hláška způsobuje zvlášť dlouhý výpis v logu jobu, takž přes parsovací utility zůstává poněkud nepřehledný. Uživatel tak použije plugin aby prohlédl složky testů a všimne si že ve dvou ze selhávajících testů je více změněných řádků než v jiných. Po prohlédnutí těchto konkrétních testů je nalezena druhá obfuskovaná chyba v konkrétním nastavení změněné komponenty.
- Po opravě chyby Test nevykazuje žádné chybové hlášení. Pro jistotu jsou soubory zkontrolovány a porovnány vůči stavu před změnou která chybu do dat zavedla. Několik souborů vykazuje změnu v generovaném výstupu se stejnou konfigurací.

Tyto konkrétní změny jsou poté osobně ověřeny testovacím týmem, a rozpoznány jako žádané změny chybného updatu.

Z těchto use cases vychází že plugin potřebuje intuitivní a rychlý systém navigace mezi složkami a musí nabízet informace o změnách v každé úrovni souborového systému. Tyto informace by pokud možno neměly sloužit pouze k detekci chyby v jinak čistém prostředí, ale také odlišování chyb a anomálií od sebe navzájem.

Je zde vidět konflikt mezi přehlcením informacemi, které v takovémto testování ve velkém běžně nastává, a nedostatkem informací k tomu aby šlo od sebe typy anomálií odlišit.

5 IMPLEMENTACE

V této kapitole vysvětluji jak funguje nastavení a výstupy pluginu. Nejedná se o vysvětlení kódu, k tomu slouží dokumentace generovaná systémem javadoc která bude přiložena k elektronické kopii této práce. Toto také považujte i za uživatelskou dokumentaci pluginu, protože ta by (až na první dvě podkapitoly) obsahovala téměř přesně stejné informace.

Pro začátek ujasním že plugin v základu neukládá nikam jaké rozdíly v souborech vlastně jsou. Ukládá pouze **počty** těchto rozdílů. Diff mezi konkrétními soubory lze vždy dostat na požádání, a hlavní účel pluginu je sledování změn, ne vytváření .patch souborů.

Možnost ukládat .patch soubor je k dispozici pouze pro build step který plugin nabízí, ale je to spíše boční funkcionalita, a nebyl na ni kladen během vývoje a návrhu důraz.

5.1 Použité extension pointy

První problém k vyřešení je kde napojit plugin na jenkins.

5.1.1 Pro vytváření diffu během buildu

Jednodušší část je automatické tvoření diffů během buildů. Jenkins nabízí několik Extension pointů pro definování nového kroku v buildu. Všechny jsou potomci třídy 'BuildStep', která sama není extension pointem ale definuje prvky společné pro všechny typy kroků. To že diff bude prováděn na konci buildu vede k jejímu potomku 'Publisher' který slouží k definici post-build akcí. 'Publisher' má dvě podtřídy, 'Notifier' a 'Recorder'. 'Notifier' slouží ke komunikaci s externími systémy, a 'Recorder' který slouží pro sběr statistik o buildu. Diffy jsou typem statistiky, takže 'Recorder' je relativně jasnou volbou pro tento plugin.

Plugin samotný také slouží jako svoje dokumentace, všechny nastavení mají k sobě připojený help text který vysvětluje jejich použití (modré otazníky na screenshotech).

5.1.2 Pro vytváření diffu na požádání

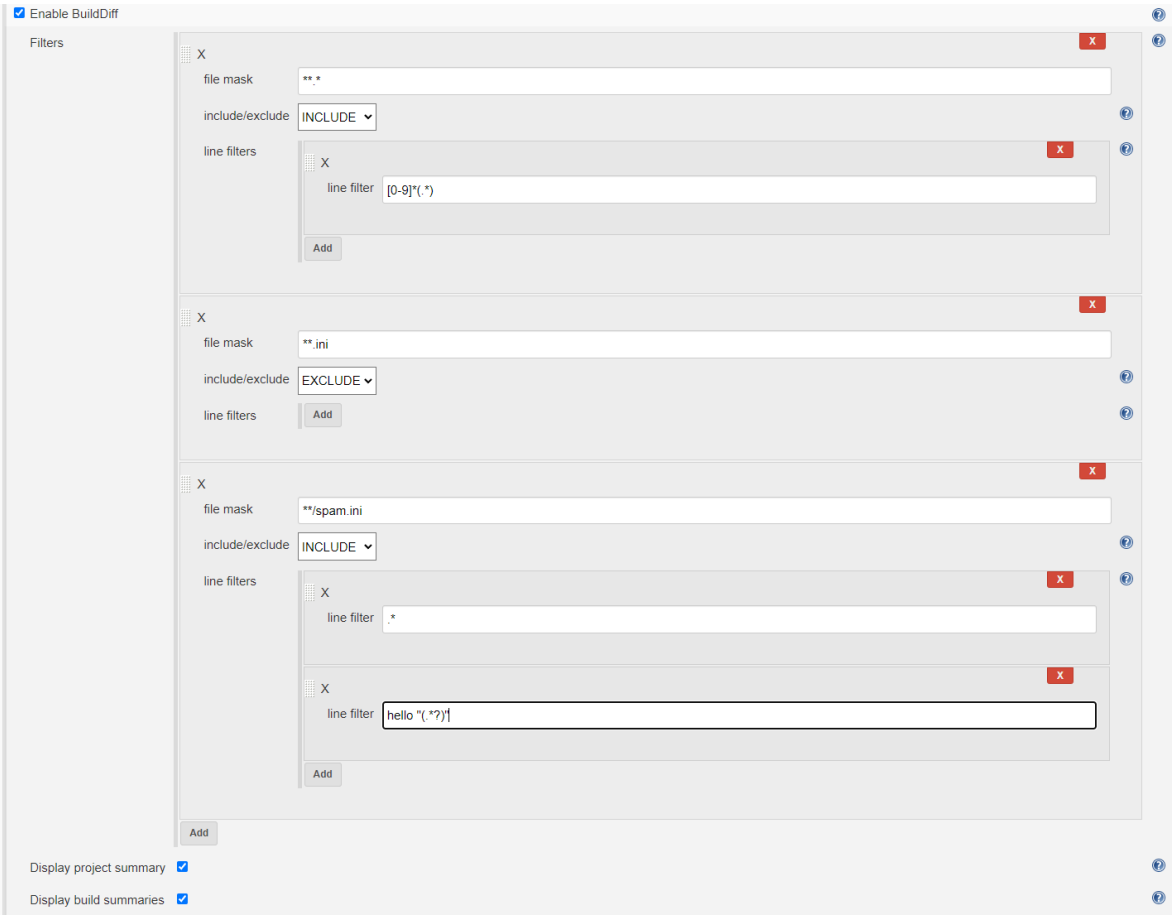
To ale plně nepokrývá use cases kdy chceme mít buildy na požádání, implementace pouze s tímto extension pointem by byla v tomto ohledu celkem restriktivní. 'Recorder' sice má metodu getProjectAction kterou lze použít k přidání funkce do projektu, ale umožňoval by vytvořit diff na požádání jen tehdy kdy by se tvořily diffy pro každý build zkrze test step.

K přidávání možností do Buildů a Jobů mohou sloužit Extension pointy 'TransientActionFactory', 'TransientBuildActionFactory', 'TransientProjectActionFactory'. Ty slouží k přidávání funkcionalit ke všem Runům a Jobům, ty ale neumožňují nastavení pro každý job zvlášť, a dodávjí 'transient' akce. Tyto objekty vznikají a zanikají společně se webovou stránkou na které se nacházejí, což znamená že v sobě nemohou uchovávat informace. Diffy by měly být uchovávány protože u aplikací na který je plugin určen bude jejich zhotovení nějaký čas trvat.

Zbývá tedy třída JobProperty, která slouží pro definování vlastností Jobu v jeho nastavení.

5.2 Použití jako vlastnost Jobu

Použití jako vlastnost jobu je možné ve vrchní sekci nastavení jobu. Vytvořil jsem toto rozhraní pro nastavení tvorby diffu



Obrázek 5.1 JobProperty menu

Jak je vidět, systém filtrů potřebuje trochu vysvětlování.

5.2.1 Souborový filter

Souborové filtry jsou definovány v klasickém unixovém formátu používaným nástrojem glob. Uživatel může definovat filtry typu 'zahrň' a 'vyčleň'. V základu není zahrnut do procesu žádný soubor. Vrchní filtr tedy definuje že mají být zahrnuty všechny soubory. Řádkový filtr budu vysvětlen v příští kapitole, takže ho prozatím ignorujte.

Prostřední filtr odpovídá všem souborům s příponou '.ini', ale je typu 'vyčleň', takže odstraňuje ze seznamu zpracovávaných souborů všechny ini soubory. Filtry které jsou níž v pořadí mají vyšší prioritu a přepisují pravidlo výše umístěných filtrů, takže toto přebíjí vrchní pravidlo o použití všech souborů

Pravidlo spodního filtru odpovídá souborům pojmenovaných spam.ini, ať už jsou kdekoliv ve struktuře složek. Je typu 'zahrň' a pod středním pravidlem, takže přebíjí

vyčlenění všech ini souborů.

Celé nastavení tedy znamená "Porovnej všechny soubory až na .ini soubory, ale chci porovnat soubory spam.ini". Tímto systémem tak jde systematicky tvořit výjimky pro výjimky a poradit si s jakkoliv specifickým požadavkem.

5.2.2 Řádkový filter

Řádkové filtry fungují jiným způsobem. Jsou zadávány pomocí regex výrazů, a specificky jsou v nich důležíte jejich capture groups.

Řádkový filtr zkouší jestli regex výraz odpovídá každému řádku. Pokud řádku odpovídá, neznamená to že je řádek porovnáván nebo není. Místo toho jsou porovnávány právě ty znaky které spadají dovnitř nějaké capture group, a znaky které se nacházejí mimo capture group jsou ignorovány. Je tedy možné porovnávat jen část řádku, a specificky ignorovat část o které je předem známo že se mění, ale jejich změny nás nezajímají.

Řádky které neodpovídají žádnému řádkovému filtru budou zahrnuty celé, regex "(.*)"který se objeví v řádkovém filtru kned po jeho vytvoření je tedy ekvivalentní nepřítomnosti řádkového filtru. Doporučuji ho nikdy nevyužívat, je to jen zbytečné plýtvání strojovým časem. Naproti tomu regex ".*"vyloučí všechny řádky.

Řádkové filtry v souborovém filtru typu 'vyluč' nemají žádnou funkci.

Stejně jako u souborového filtru, řádkové filtry umístěny níže mají výšší prioritu než ty umístění výše.

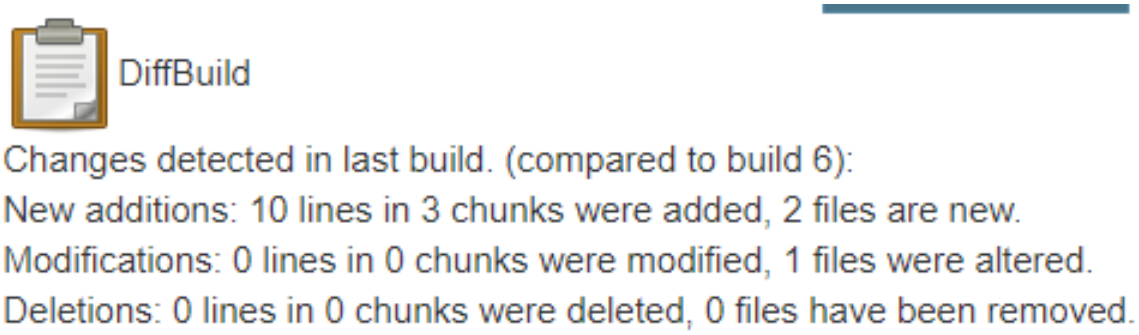
Řádkový filtr v prvním souborovém filtru tedy značí že ve včech souborech je ignorována sekvence čísel na začátku řádku.

Řádkové filtry ve spodním souborovém filtru znamenají že jsou plně ignorovány všechny řádky kromě těch které začínají slovem "hello"za kterým následuje string umístěný v uvozovkách. Umístění capture group v tomto regexu společně s prioritou a prvním filtrem znamená, že z celého souboru jsou porovnávány pouze obsahy těchto uvozovek. Uživatel značí že změny v čehokoliv jiném ho nezajímají.

Používat regex na každý řádek je celkem výkonově náročné, takže doporučuji je použít ve souborovém filtru který bude odpovídat právě jen ty změny které chceme ignorovat.

5.2.3 Reporty

Poslední dvě možnosti vypínají nebo zapínají reporty o diffech na stránce jobu a buildu. Na obou místech mají stejnou podobu.



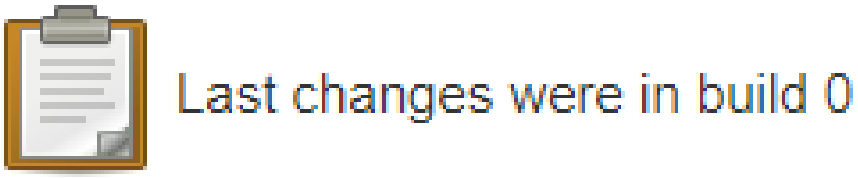
Obrázek 5.2 Report se změnami

Pokud byly změny v tomto buildu (na stránce buildu) nebo v posledním buildu (na stránce jobu) tak zobrazují množství změn.

'Chunky' jsou nepřetržité série řádek které byly změněny. Chunk je považován za přidán nebo odebrán pouze když všechny řádky v něm byly přidány nebo odebrány, změna v řádku způsobí že chunk je vyhodnocen jako změna. To může skončit s 'falešnou' změnou pokud je některá skupina řádku přídána nebo odebrána a druhá má opačnou změnu hned vedle nich.

Informace o souborech a řádcích fungují přesně dle očekávání.

Pokud nejsou v buildu nabo posledním buildu žádné změny, report vypadá takto:



Obrázek 5.3 Report beze změn

V případě, že report tvrdí že poslední změna byla v buildu "0"(jako zde), je význam ten, že žádném buildu nebyla změna. Jinak to bude číslo posledního buildu se změnou.

U jobu jsou tyto reporty umístěny vpravo mezi informacemi o stavu jobu jako celku. U buildu jsou uprostřed mezi ostatními reporty o buildu.

Za zmínku stojí že tyto reporty nebudou zaručeně korektní pokud není používán diffovací step. Zobrazují informace v závislosti na tom jestli diffy skutečně existují. Pokud budou diffy vytvořeny na požádání, tyto informace se aktualizují, ale uživatel by se na ně neměl stoprocentně spoléhat když nepoužíva přidružený step.

5.3 Použití jako step buildu

První věc: step nelze používat bez nastavené job property, step neudělá nic a selže. Nejedná so o nečekanou chybu, step kontroluje Job na přitomnost BuildDiffJobProperty a selhává regulérním způsobem. Je to tak z toho důvodu že mi nepřišlo že mít filter nastavení zvlášť pro step a pro property by byl dobrý design. Nepodařilo se mi přijít na způsob jak step znepřístupnit když property není nastavená.

S tím z cesty, nastavení stepu má tento interface:

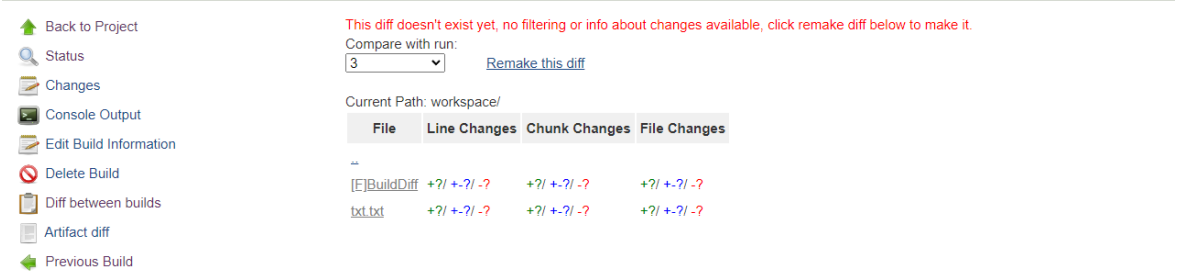
Obrázek 5.4 Step menu

Možnost 'Save diff files' umožňuje uložit do artefaktů všechny diffy které jsou kalkulovány ve formátu .patch. Toto je pro případ že by s nimi chtěl uživatel dále pracovat. Soubory .patch jsou ve stejné struktuře jako původní artefakty. Za zmínku stojí to že tato složka by pravděpodobně měla být vyloučena z diffu pomocí souborového filtru, jinak nastane mezi buildy její rekurzivní růst.

Druhá možnost určuje jaký je výsledek stepu při nalezení změn. Bez nalezení změn je výsledek vždy "Success".

5.4 File manager

Po rozkliknutí možnosti BuildDiff v levém menu na stránce buildu se uživatel dostane do file manageru, který v základu bude vypadat nějak takto



Obrázek 5.5 Filemanager bez diffu

Jak varuje červený text nahoře, tento diff ještě nebyl vytvořen. Uživatel uvidí soubory a složky v artefaktech momentálního buildu. Všechny informační sloupce mají v sobě otazníky, protože status je neznámý.

Podsložky vždy začínají symboly

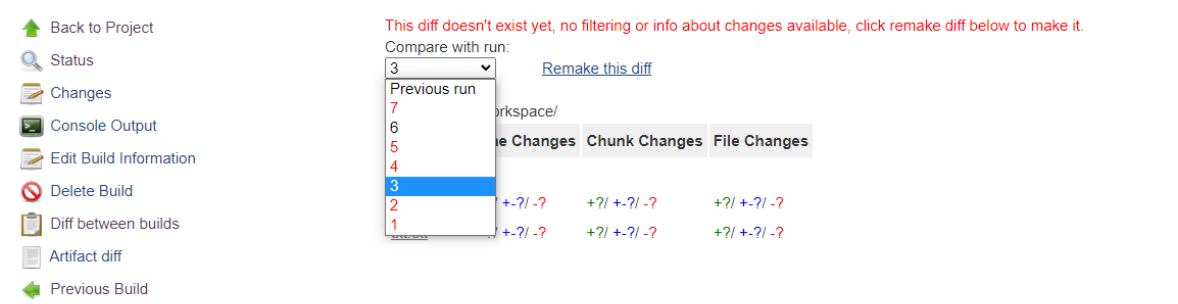
$$f$$

a bývají na prvních pozicích v seznamu souborů.

Selektor nad seznamem souborů umožňuje vybrat se kterým buildem momentálně porovnáváme.

Na kterékoliv místo v souborovém přístupu lze přistoupit zkrze URL adresu následujícího formátu.

<adresa momentálního runu>/builddiff/<číslo runu k porovnání>/<cesta k souboru nebo ke složce>



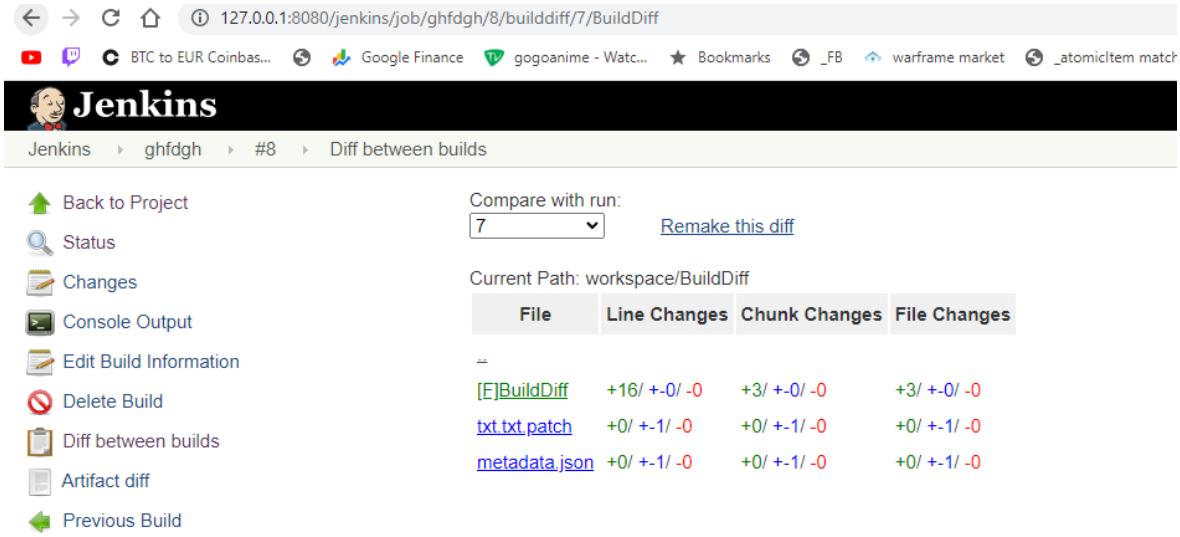
Obrázek 5.6 Výběr buildu

Po rozkliknutí se ukáže volba buildu se kterým porovnáváme. K dispozici jsou předchozí i následující buildy. Buildy pro které nebyl diff ještě vytvořen jsou psány v červené barvě. Pro pohodlnost je vždy k dispozici volba "poslední build", pro případ že uživatel nechce přemýšlet o tom, na který build se momentálně dívá, a který je poslední.

Pro změnu momentálního buildu lze použít odkazy v levém menu, které jsou definovány Jankinsem samotným.

Pro vytvoření diffu poklikáme na odkaz "Remake this diff"napravo od selektoru. V závislosti na množství artefaktů tvorba diffu může trvat až několik minut.

S vytvořeným diffem uvidíme množství rozdílů mezi soubory a složkami.



Obrázek 5.7 Filemanager s diffem

Informace o změnách fungují stejným způsobem jako v reportech. Pro složky jsou tyto hodnoty sumou všech změn v jejich podsouborech a podsložkách.

Rozkliknutím souboru se dostaneme do porovnání souborů. Momentálně je v pluginu mírně obtěžující polochyba, která způsobuje že když je ve složce soubor a podložka se stejným jménem, file manager uživatele vždy přesune do stránky se složkou i když klikne na soubor.

Pro mitigaci tohoto problému se v těchto případech pod selektorem buildu objeví odkaz který umožňuje přepnout mezi stránkami se složkou a souborem. Pokud se chce uživatel dostat k jednomu nebo druhému pomocí URL, stačí k adrese přidat GET parametr "?fileType=file"pro soubor a "?fileType=directory"(nebo žádný parametr) pro složku.

Obrázek 5.8 Diff souboru

Tato stránka je relativně sebevysvětlující, jedná so o porovnání souborů v GNU unified diff formátu.

5.5 Rozbor nedostatků nalezených během testování a používání

5.5.1 Využití paměti

Plugin udržuje informace o počtu změn pro každý soubor a složku v paměti počítače. To znamená 9 proměnných typu int pro každý soubor a složku pro každý build pro který byl diff vytvořen. Pro extrémní množství souborů by toto mohl být problém, ale jinak by měly paměťové požadavky pluginu být v rámci jednotek až desítek MB paměti. Např. 20 buildů na 50 jobech, každý po 1000 souborech, každý soubor udržuje 9 intů a 2 reference na artefaktové objekty každá po 8 bytech. Plugin bude potřebovat 20*50*1000*11*8= 88 Mbytů

Plugin by tyto informace mohl uchovávat v souborech, ale to by vyžadovalo náročnější implementaci.

5.5.2 Serializace

Nápodobně jsou tyto informace ukládaný v XML formátu když jsou data jenkinsu serializovány. Každý job je ukládán do jednoho XML souboru To znamená že tyto XML soubory budou bobtnat co se týče velikosti, a nebudou čitelné člověkem. Toto by mohl být větší problém než RAM, kvůli inherentně neúspornosti formátu XML pro ukládání velkého množství čísel.

Problém by bylo potřeba vyřešit jinou formou serializace, např. uchovávání dat na disku v binární formě.

5.5.3 Omezení na artefakty

Plugin nepracuje s ničím jiným než artefakty. Práce s jiným externím systémem nebyla jedním z use case,

5.5.4 Nebezpečí přetížení serveru při kalkulaci rozdílů mezi velkými souborů

Při porovnávání velmi velkých souborů (desítky megabytů) je možné že Jenkins serveru dojde paměť. Uživatel si tedy musí dávat pozor na definici svých souborových filtrů pokud takové soubory ukládá mezi artefakty. Plugin není určený k porovnávání binárních souborů, a na takto velké soubory prostě není stavěný.

Pro částečné řešení tohoto problému by šla omezit maximální velikost souboru který se bude plugin snažit zpracovat.

ZÁVĚR

Cílem práce bylo naimplementovat plugin do CI systému Jenkins, který by sledoval změny ve výstupech opakovaně prováděných akcí a přehledným způsobem je prezentoval ve webovém rozhraní tohoto systému.

Plugin byl úspěšne implementován a zprovozněn, přestože v jeho implementaci zůstává několik technických nedostatků

Tyto technické problémy omezují na jaké škále může být plugin používán. Základní implementace uchovávání informací o buildech v Jenkinsu není určena pro velké objemy dat, a neklade tak velký důraz na spoření místem na disku a množstvím používané pamětí za běhu. Flexibilita systému Jenkins umožňuje tyto problémy řešit, ale vyšší prioritou byly doposud systémy určené pro filtrování porovnávaných dat a pro jejich přehlednou prezentaci, jejichž implementací a rozšiřováním byla doposud strávena většina vývojového času.

Téma možné budoucí publikace pluginu na veřejný repozitář Jenkinsu zůstává nejisté. Původně bylo zamýšleno že by byl plugin veřejně dostupný, ale ve stávajícím stavu je jeho používání mírně riskantní, protože špatné nastavení souborových filtrů může způsobovat pád Jenkins serveru. Druhý problém je že se nedávno zpřísnila politika ohledně intelektuálních vlastnictví firmy, a na plugin by musel být vypracován posudek povolanou osobou aby jsem obdržel povolení ho publikovat.

Přes tyto nedostatky je plugin nasazený v praxi už několik měsíců, a svůj hlavní účel podpory regresního testování vykonává úspěšně.

SEZNAM POUŽITÉ LITERATURY

[1] Spillner, A.; Linz, T.; Schaefer, H.: *Software testing foundations*. Santa Barbara, CA: Rocky Nook, třetí vydání, Leden 2011.

[2] Component Testing Tutorial: A Comprehensive Guide With Examples and Best Practices. [Online; navštíveno 24.05.2023].
URL `https://www.lambdatest.com/learning-hub/component-testing`

[3] Ramesh Gopalaswam, S. D.: *Chapter 6, System and Acceptance Testing*. ISBN 9788177581218.

[4] Belmont, J.-M.: *Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI, 1st Edition, Kindle Edition*. Packt Publishing, 2018, ISBN 9781789130485.

[5] Travis CI online documentation. [Online; navštíveno 14.05.2023].
URL `https://docs.travis-ci.com/`

[6] Aquasec Travis CI security report. [Online; navštíveno 08.05.2023].
URL `https://blog.aquasec.com/travis-ci-security`

[7] Github statistics about CI usage. [Online; navštíveno 08.05.2023].
URL `https://github.blog/2017-11-07-github-welcomes-all-ci-tools/`

[8] Circle CI online documentation. [Online; navštíveno 15.05.2023].
URL `https://circleci.com/docs/`

[9] Jenkins plugin reposirory. [Online; navštíveno 18.05.2023].
URL `https://plugins.jenkins.io/`

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CI Centinous Integration

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH