Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Využití virtualizace při přípravě infrastruktury pro testování webových aplikací

Radka Nepejchalová

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Radka Nepejchalová

Informatika

Název práce

Využití virtualizace při přípravě infrastruktury pro testování webových aplikací

Název anglicky

Virtualization utilization for automatic infrastructure creation for web application testing

---

Cíle práce

Diplomová práce se zabývá možnostmi využití virtualizace v prostředí vývoje webových aplikací a to především v oblasti automatického deploymentu pro automatické testy. Cílem je získat přehled o výhodách a možnostech použití těchto metod a vyhodnotit jejich přínosy.

Metodika

Metodika diplomové práce bude založena na studiu, analýze a syntéze odborných publikací. Získané poznatky budou využity při tvorbě vlastního řešení pro konkrétně definovaný problém.

Doporučený rozsah práce

60 stran

Klíčová slova

virtualizace automatický vývojová konfigurace řízení

Doporučené zdroje informací

HASHIMOTO, Mitchell. Vagrant: up and running. xiii, 138 p. ISBN 9781449335830.

HUMBLE, Jez a David FARLEY. Continuous delivery: reliable software releases through build, test, and deployment automation. Upper Saddle River, NJ: Addison-Wesley, 2010, xxxiii, 463 p. ISBN 9780321601919.

NEMETH, Evi. UNIX and Linux system administration handbook. 4th ed. Upper Saddle River, NJ: Prentice Hall, c2011, xlvii, 1279 p. ISBN 0131480057.

TOSATTO, Daniele. Citrix XenServer 6.0 administration essential guide: deploy and manage XenServer in your enterprise to create, integrate, manage, and automate a virtual datacenter quickly and easily. Birmingham: Packt Pub., 2012, v, 346 p. ISBN 978-1849686167.

Předběžný termín obhajoby

2015/16 LS – PEF

Vedoucí práce

Ing. Alexandr Vasilenko

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 19. 11. 2015

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 11. 2015

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 12. 02. 2016

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Využití virtualizace při přípravě infrastruktury pro testování webových aplikací“ jsem vypracovala samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autorka uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušila autorská práva třetích osob.

V Praze dne 15. 3. 2016 _____

Poděkování

Ráda bych touto cestou poděkovala vedoucímu práce Ing. Alexandru Vasilenkovi za jeho průběžné rady a připomínky.

Využití virtualizace při přípravě infrastruktury pro testování webových aplikací

Virtualization utilization for automatic infrastructure creation for web application testing

Souhrn

Práce si klade za cíl přiblížit některé ze současně využívaných postupů či principů v rámci vývoje softwaru. V dnešní době je kladen velký důraz na efektivitu a kvalitu spolupráce týmů. Právě zde vzniká potřeba řídit se principy kontinuální integrace a to především u komplexnějších projektů, na kterých aktivně spolupracuje paralelně více vývojářů. Je zde větší potřeba integrovat přibývající nový kód a ujistit se, že je vzniklý celek funkční. Jedním ze základních kamenů kontinuální integrace je pravidelné testování - tedy odhalování problémů, které by mohly nastat při nasazení v produkčním prostředí. Takové testování však může být velice nákladné a ke snížení těchto nákladů vede právě využití virtualizace. Testování nových verzí aplikace je díky virtualizaci velice snadné. Každý vývojář si může na vlastním počítači vytvořit vlastní virtuální stanici a na ní nanečisto vyzkoušet běh aplikace. Část práce se věnuje právě těmto zmíněným postupům a logicky je propojuje. Praktická část se pak věnuje demonstraci řešení reálného problému.

Summary

The main aim of this thesis is to approximate some of the currently used techniques and principles within software development. Nowadays, great emphasis is placed on efficiency and quality of teams. Adopting the principles of continuous integration brings multiple benefits, especially for complex projects where many developers work at parallel. There is a greater need to integrate increasing generated code and make sure that the newly formed unit keeps functional. One of the main principles of the continuous integration is regular testing possibly revealing the problems that might occur during production deployment. However, such testing can be very expensive and using virtualization can be the key to reduce these costs. Testing new versions of applications can be made very easily using virtualization. Part of the work is devoted to all these previously mentioned parts. The practical part is then devoted to demonstrate solutions of real problems.

Klíčová slova

kontinuální integrace, vývoj software, testování software, virtualizace

Keywords

continual integration, software developement, software testing, virtualization

Obsah

1 Úvod

V dnešní době jsou dostupné aplikace ať již webové nebo mobilní na cokoli člověka napadne. Tyto aplikace jsou více či méně složité, ale vždy za nimi stojí tým vývojářů. Pod tímto týmem si většina lidí představí pouze programátory. Při změně zaměstnání jsem se jakožto neprogramátor stala součástí vývojářského týmu a teprve tehdy jsem si uvědomila, co všechno s sebou vývoj SW nese a kolik různých profesí a úkolů je v tomto odvětví potřeba stále řešit.

Celé IT odvětví včetně právě vývoje SW se jako řada jiných vyvíjí velice rychle a jde zde obrovská konkurence. Týmy se snaží svoji práci zefektivnit a zkvalitnit a vznikají tak soubory pravidel či principů, které jsou tvořeny profesionály s mnohaletou praxí v oboru. Vzniká také mnoho komerčních technologií i nástrojů vyvíjených komunitou, které vývojářům zjednodušují práci. Vývoj takových technologií je velice náročné stíhat, protože každý rok je jich na trh uvedeno obrovské množství a mnoho z nich je možné objevit až při výskytu konkrétního problému.

Motivací k výběru tématu práce byla tedy především reálná potřeba řešit konkrétní úkol v rámci vlastního zaměstnání. V průběhu tohoto řešení jsem narazila na mnoho překážek, ale také výzev. Mnohem více jsem pronikla do nástrojů virtualizace a mimo jiné jsem se také leccos naučila o zajímavých a moderních technologiích jako např. Docker, které jsem do té doby neznala a které mnoho z těchto problému velice účinně řešily.

V rámci diplomové práce není prostor pro kompletní pokrytí všech částí řešeného problému a práce samotná v podstatě slouží jako jakýsi přehled možných řešení a úvod do jejich problematiky.

2 Cíl práce a metodika

Cílem práce je analýza využití virtualizačních technologií v prostředí vývoje webových aplikací a to v oblasti vytvoření infrastruktury pro automatické nasazení v rámci automatického testování.

Dílčím cílem práce je vymezení některých pojmů ve vývoji SW a zdůraznění jejich důležitosti a užitku v celém procesu.

Práce se v teoretické části detailněji věnuje principům kontinuální integrace, testování SW a poté také základům virtualizace.

Praktická část pak pomocí vlastního řešení zadaného úkolu demonstruje propojení všech tří částí řešených v rámci teoretické části.

Informace obsažené v práci jsou čerpány v první řadě z odborných publikací a literatury. Jako druhotné zdroje jsou zde použity informace z webových prezentací a odborných článků.

3 Přehled řešené problematiky

Celé IT odvětví se rozvíjí obrovskou rychlostí a pro zachování konkurenceschopnosti v jakémkoli z nich se firmy snaží veškerou práci zefektivnit a zautomatizovat.

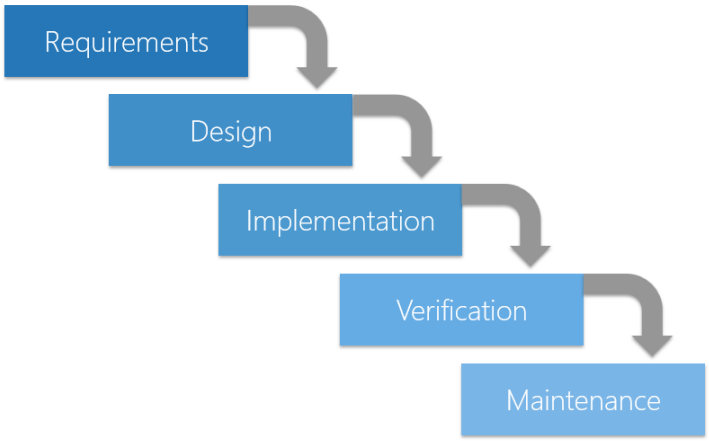
Ve firmách, kde probíhá vývoj komplexních softwarových produktů v týmu, je snaha práci týmu koordinovat, aby byl výsledný produkt funkční a zároveň splňoval daná kritéria. V průběhu let byla odborníky nebo firmami s dlouholetými zkušenostmi vyvinuta některá doporučení či principy jak právě práci v týmech usnadnit, zrychlit a tím i zlevnit. Jedním z takových souhrnů různých vývojářských nástrojů a metod je kontinuální integrace.

3.1 Kontinuální integrace

Při vývoji softwarového projektu vývojáři pracují samostatně na dílčích úkolech v rámci projektu, které je poté nutné propojit. Tento proces se označuje jako integrace.

Kontinuální integrace (CI), tedy soustavné přidávání nových částí do programu, se vyvinula jako součást vývoje softwaru z různých důvodů (Zamborský, 2012) .

Tradiční metodiky vývoje software, jako je například vodopádový model, zařazují fáze integrace, testování a kontroly kvality až po dokončení fáze vývoje. Odhad doby samotného vývoje je díky neurčitosti trvání doby integrační fáze a fáze kontroly kvality prakticky nemožný (Hujer, 2012).



Obrázek 1 - Vodopádový model vývoje softwaru (pepgotesting.com)

Potřeba řídit se principy kontinuální integrace vzniká především u komplexnějších projektů, na kterých aktivně spolupracuje paralelně více vývojářů. Je zde větší potřeba

integrovat přibývající vzniklý kód a ujistit se, že je nově vzniklý celek funkční. Samotným pravidelným ověřováním funkčnosti celku je možné pověřit přímo vývojáře, ale v ten moment se vývojáři zabývají činností, která nemá přímou hodnotu. Bez kontinuální integrace by navíc této činnosti přímou úměrou s růstem týmu přibývalo.

Současně ale s touto činností není možné čekat až na konec projektu. Takový postup by vedl k mnoha problémům - zhoršení očekávané kvality kódu nebo např. k prodražení projektu kvůli zpoždění. Cílem kontinuální integrace je tedy řešit tyto problémy hned jakmile nastanou a po menších částech.

Martin Fowler, autor blogu mapující trendy ve vývoji softwaru, definuje kontinuální integraci takto:

„Kontinuální integrace je soubor postupů ve vývoji software, kde členové týmu integrují svoji práci často, obvykle každá osoba alespoň jednou denně - což vede k mnoha integracím denně. Každá integrace je ověřena automatickým sestavením - tzv. buildem (včetně testů) k detekci případných chyb tak rychle, jak je to jen možné. Mnoho týmů zjišťuje, že tento přístup vede k výraznému snížení integračních problémů a umožňuje týmu rychleji vyvíjet soudržnější software.“ (Fowler, 2006)

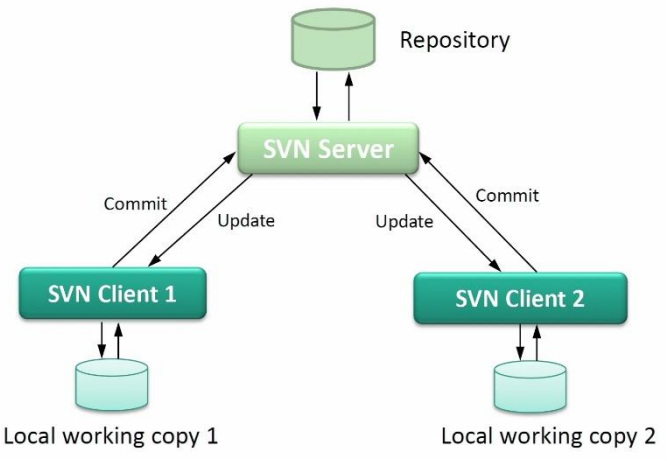
„Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.“ (Fowler, 2006)

3.1.1 Klíčové principy kontinuální integrace

Zdroje zabývající se kontinuální integrací se v jádru shodují na některých společných principech, v detailech se ale mohou lišit. Níže uvádím nejčastěji citované klíčové principy CI podle Martina Fowlera (Fowler, 2006).

3.1.1.1 Centrální úložiště zdrojového kódu

Jedním ze základních stavebních kamenů kontinuální integrace je nutnost používání centrálního úložiště zdrojového kódu. Každý softwarový produkt obsahuje velké množství souborů rozličných typů – soubory se zdrojovým kódem, konfigurační soubory, knihovny třetích stran nebo grafické soubory jako ikony či obrázky. V případě spolupráce více vývojářů na jednom produktu, je potřeba zajistit, aby všichni stále pracovali s aktuálními verzemi těchto souborů. V minulosti bylo běžné ke sdílení dat použití síťových disků. Toto řešení se však rychle ukázalo jako nevyhovující a to především kvůli absenci možnosti procházet historii souborů se zdrojovým kódem (Hujer, 2012). Systémy pro správu zdrojového kódu (Source Code Management), jinak nazývané také systémy verzování (Version Control), jsou dnes již běžnou a nedílnou součástí velké většiny softwarových projektů.



Obrázek 2 - Princip práce s verzovacím systémem Subversion (Wikimedia Commons, 2011)

Mezi nejznámější verzovací systémy patří open-source systém Apache Subversion a Git (využívaný také pro verzování jádra operačního systému Linux).

Jakmile je týmem používán verzovací systém, je nezbytné, aby byly do verzovacího systému ukládány vždy všechny soubory, které jsou potřeba k úspěšnému sestavení aplikace, ale i takové, které mohou dalším vývojářům usnadnit práci jako např. konfigurační soubory vývojového prostředí (Hujer, 2012). Dle Martina Fowlera (Fowler, 2006) stačí dodržet jednoduché pravidlo – každý by měl být schopen provést úspěšné sestavení na čistě nainstalovaném počítači pouhým „check-outem“ projektu z verzovacího systému.

Jednou z hlavních funkcí verzovacích systému je také možnost vytváření více tzv. větví vývoje (branches) např. pro současnou práci na různých projektech, verzích či oddělených funkcionalitách projektu. Tato vlastnost je užitečná, ale je s ní třeba zacházet s obezřetností a především si dát pozor na nadužívání, které by komplikovalo pozdější integraci.

Mimo tyto větve zde ale především existuje hlavní větev – mainline, ve které by měla být prováděna většina práce a případné jiné větve jsou do ní po dokončení vývoje reintegrovány (Fowler, 2006).

3.1.1.2 Automatizace buildu (sestavení)

Proces sestavení funkční aplikace může být často velice složitý. Může zahrnovat spoustu mezikroků jako kompilace vlastního zdrojového kódu, kopírování souborů, práci s databázemi apod. Tyto úkoly by mohly být ve většině případů zautomatizovány. Automatizace přináší nejen zrychlení práce, ale také mnohem menší náchylnost k chybám obvykle způsobených zapojením lidského faktoru do procesu.

Nástrojů pro sestavování softwarových aplikací existuje celá řada. Jedním z nejznámějších je v operačních systémech Unix a Linux již léty prověřený make (Free Software Foundation). Pro projekty v jazyce Java je často používaný Apache Ant či alternativně Maven nebo další v současnosti velice populární – Gradle.

Častou chybou při automatizaci je, že ne všechno je obsahem automatického sestavení – buildu. Martin Fowler přidává v tomto ohledu pravidlo, ve kterém říká, že každý by měl být schopen na zcela „čisté“ pracovní stanici pomocí check-outu zdrojových souborů z verzovacího systému a jednoho příkazu získat funkční aplikaci (Fowler, 2006).

„I'll elaborate my earlier rule of thumb: anyone should be able to bring in a virgin machine, check the sources out of the repository, issue a single command, and have a running system on their machine.“ (Fowler, 2006)

Jednou z dalších funkcí nástroje pro sestavování softwarových aplikací by mělo být také hlídání změn. Některá větší sestavení – buildy – trvají dlouho a není možné ani žádoucí jejich sestavení provést při každé sebemenší změně. Toto již často řeší server kontinuální integrace, který v určených intervalech kontroluje změny ve verzovacím systému a build spouští pouze v případě, že zde změny proběhly a byly zároveň vyhodnoceny jako významné.

3.1.1.3 Automatizace testování

V tradičním pojetí sestavení zahrnuje kompilaci kódu a spojení s dalšími součástmi, tak aby bylo možné získat spustitelnou aplikaci. Aplikace ale může být spustitelná a přesto to

neznamená, že funguje korektně. Moderní staticky typované programovací jazyky jsou schopny mnoho chyb odfiltrovat již při kompilaci, ale mnoho z nich přesto unikne. Z tohoto důvodu je velice výhodné zakomponovat automatické testy přímo do procesu sestavení aplikace (Fowler, 2006). Pro samo otestování kódu je ovšem potřeba vytvoření sady automatických testů pokrývajících, co největší množství kódu. Množství chyb a problémů v aplikaci je tím opět možné zredukovat.

3.1.1.4 Časté commity nového kódu vývojářem

Jedním ze základních principů integrace je komunikace. Integrace pomáhá vývojářům informovat ostatní vývojáře o změnách, které právě provedli (commit), a tím zrychlit a zefektivnit vlastní vývoj (Fowler, 2006).

Předpokladem pro commit do vývojové větve je vývojářem ověřená možnost sestavení na vlastní lokální kopii. S tím souvisí i nutnost pravidelné synchronizace lokální kopie s aktuální verzí kódu ve verzovacím systému. Díky časté synchronizaci je možné rychle odhalit konflikty mezi lokální kopií a verzí z verzovacího systému způsobenou např. paralelním vývojem více vývojářů a následně je opravit. V případě, že se na konflikty přijde po delší době, je mnohem obtížnější odhalení jejich příčiny, protože jeden commit pak obsahuje mnohem delší a větší úsek práce a tím je těžší následná oprava.

V případě, že vývojář ověří funkčnost sestavení, je již možný vlastní commit do vývojové větve.

3.1.1.5 Z každého commitu existuje build

Každodenní commitování kódu přidává do procesu výhodu často testovaného buildu, což znamená, že hlavní vývojová větev zůstává stále ve stabilním stavu. Prakticky je zde ale stále prostor pro mnoho problémů. Jedním z nich je lidský faktor – tedy, že vývojář nedodrží nutnost synchronizace s repositářem předtím než provádí commit nebo problémy způsobené rozdílnostmi samotných vývojových prostředí vývojářů.

Z těchto důvodů je vhodné zajistit pravidelné spouštění sestavování na serveru kontinuální integrace. Jen v případě, že build po commitu dopadne dobře, je možné ho považovat za kompletní. Za každý commit zároveň nese zodpovědnost vývojář, který ho provedl a měl by sledovat jeho vývoj a popř. opravit problémy.

Jak je zmíněno dříve v textu, toto je možné zajisti pomocí serveru kontinuální integrace, který pomocí tzv. pollingu hlídá nové commity a v určených intervalech tak spouští jednotlivé buildy a poté zveřejní výsledky, popř. pošle notifikaci o výsledcích autorovi commitu např. e-mailem. Server kontinuální integrace není nezbytně nutný, ale jeho použití je mnohem pohodlnější a spolehlivější než manuální proces. Na druhou stranu používání integračního softwaru vyžaduje určitou úroveň znalosti správného nastavení jeho jednotlivých komponent, údržbu a správu.

V případě manuálního procesu je postup velice podobný jako v případě sestavení na lokální kopii vývojáře. Vývojář si po vlastním commitu sedne k speciální pracovní stanici vyhrazené pouze pro integraci, stáhne aktuální verzi kódu z verzovacího serveru a spustí build. Až pokud je build dokončen správně, lze commit považovat za dokončený (Hujer, 2012).

3.1.1.6 Udržování, co nejkratší délky trvání buildu

Klíčovou podstatou kontinuální integrace je poskytování rychlé zpětné vazby vývojářům. Pro celý proces je tak nesmírně důležité, aby sestavení trvalo krátkou dobu. Podle popisu principů dle Martina Fowlera (Fowler, 2006) by sestavení nemělo trvat déle než 10 minut. V případě delší doby stoupá pravděpodobnost, že vývojáři nebudou sestavení spouštět lokálně před vlastním commitem.

„Často je sestavení zpomalováno testy, které pracují s externími zdroji, jako je databáze nebo služby. V takovém případě je možné sestavení rozdělit na dvě části – primární (spouští se po uložení změn do úložiště), sekundární (spouští se po úspěšném doběhnutí primárního). Výhoda tohoto přístupu je v tom, že vývojáři stačí, když počká na výsledek primárního sestavení. Pokud by sekundární sestavení odhalilo nějaké chyby, tak to není tak velký problém, protože testy důležité funkčnosti jsou zahrnuté v primárním sestavení.

Pokud by sekundární sestavení trvalo příliš dlouho, je možné využití například více serverů, na kterých testy poběží (integrační servery často funkčnost pro distribuované sestavení a testování obsahují). V tu chvíli musí být testy schopny běžet izolovaně. Hlubší rozbor možností paralelního spouštění testů na více strojích nicméně přesahuje rozsah této práce.“ (Hujer, 2012)

3.1.1.7 Testování buildu v prostředí podobném produkčnímu prostředí

Cílem testování je odhalení problémů, které by mohly nastat při nasazení v produkčním prostředí. Z tohoto důvodu je více než vhodné, aby se testovací prostředí podobalo co nejvíce tomu produkčnímu. Pokud jsou testy spouštěny v odlišných podmínkách, je každá taková odlišnost potenciální riziko, že chování v produkčním prostředí bude rozdílné od chování v prostředí testovacím (Fowler, 2006).

Správným přístupem by tedy bylo přizpůsobení testovacího prostředí podle produkčního. Ideální by byl čistý klon operačního systému, využití stejného databázového softwaru, stejných verzi softwaru a zachování stejných verzí knihoven. Martin Fowler ve své publikaci o kontinuální integraci (Fowler, 2006) zmiňuje také využití identického systému, stejných IP adres a portů, ale za reálných podmínek je velice těžké dodržet i předchozí body v závislosti na vyvíjené aplikaci. Přesto by zde měla ale zůstat alespoň snaha o co největší podobnost.

3.1.1.8 Snadný přístup vývojářů k buildům a k jejich výsledkům

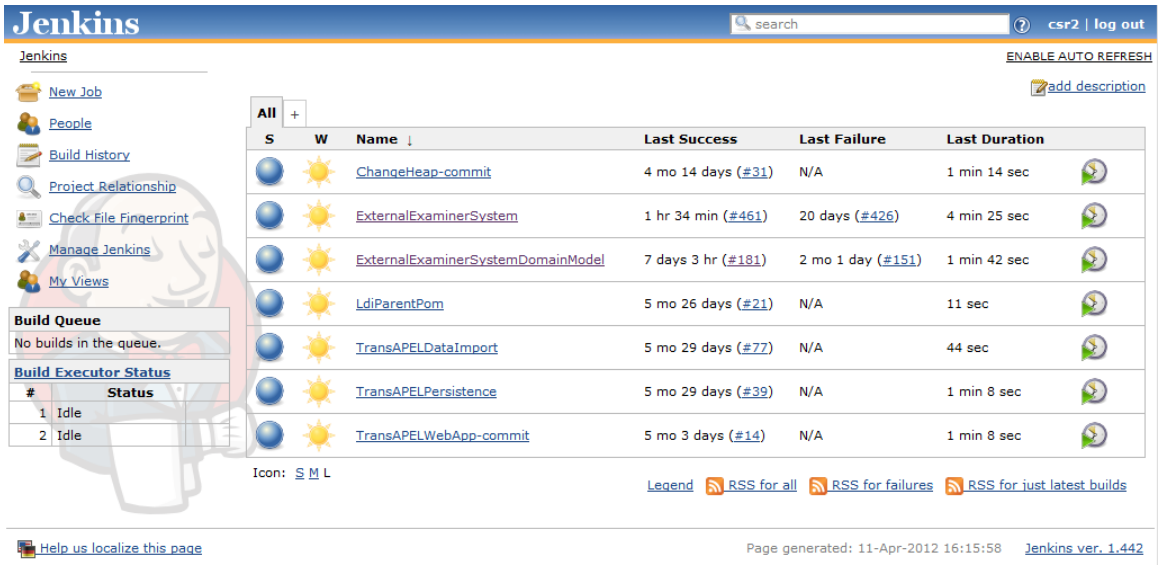
Z předchozích kroků by mělo být již jasné, že proces kontinuální integrace je především o usnadnění komunikace a přístupu k informacím, takže má každý přehled o stavu systému a všech změnách, které se v něm udály (Fowler, 2006).

Pro vývojáře je především důležité udržovat si přehled o stavu posledního buildu. Většina serverů kontinuální integrace, jako např. velice populární Jenkins CI (Jenkins, 2016), umožňuje podrobné monitorování stavů u jednotlivých úloh – jobů.

Klasické zobrazení pak vypadá jako seznam jednotlivých jobů a indikátorů jejich stavů. Tyto stavy jsou odlišeny barvami, podobně jako na semaforu (v závislosti na konkrétním nastavení serveru). Zelená znamená, že build je v pořádku, žlutá, že build proběhl, ale vyskytly se dílčí problémy – např. některé z testů neskončily jak měly a červená značí, že build z nějakého důvodu nedoběhl korektně až do konce.

Dalším často zobrazovaným údajem je také údaj o tom, jaký je trend stavů jednotlivých jobů, který je v případě Jenkins CI zobrazován pomocí ikon podobných jako u předpovědi počasí, kde slunečno je dlouhodobá stabilita jobu, kdežto bouřkový mrak představuje problémový, často nedokončený job.

Jak již bylo zmíněno dříve, zobrazení informací není jedinou předností serveru kontinuální integrace. Další výhodou, usnadňující vývojářům přístup k informacím o stavech buildů a k jejich výsledkům, je možnost automatického zasílání notifikačních zpráv, ať již e-mailových či SMS. Tyto notifikace je navíc možné nastavit tak, že jsou zaslány např. pouze v případě chybného dokončení buildu a pouze vývojářům, kteří do konkrétního buildu commitovali.



Obrázek 3 - Zobrazení Dashboard na serveru CI Jenkins

3.1.2 Shrnutí procesu CI

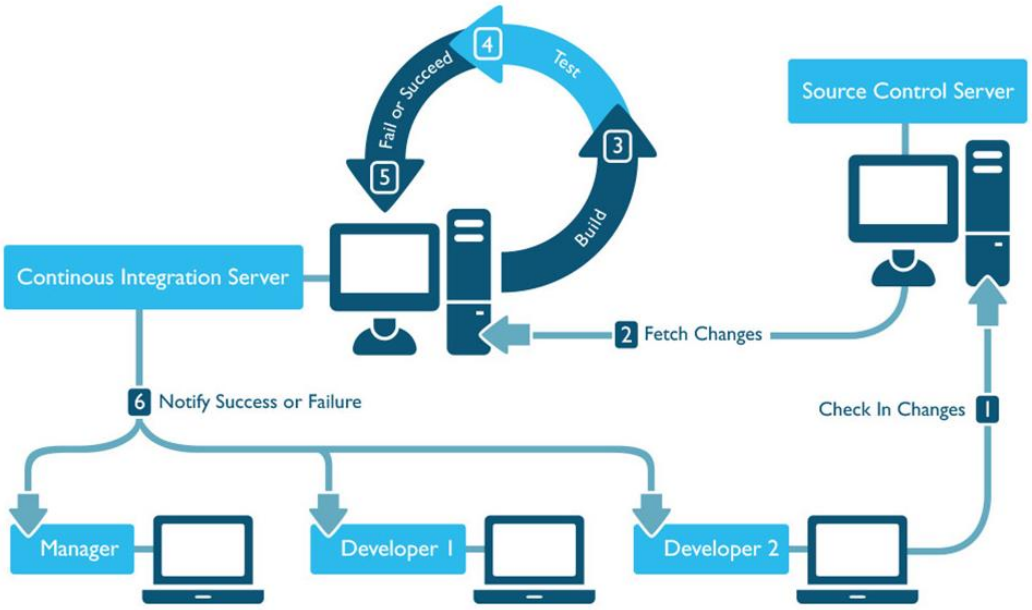
Celý proces lze popsat schématem viz Obrázek 4 a posloupností následujících kroků:

Vývojář vytvoří novou větev (feature branch) z původní master větve ve verzovacím systému, sám se ve vývojovém prostředí do této větve přepne (switch) a stáhne si z ní zdrojový kód a soubory potřebné k vývoji (checkout). Tato větev bude využívána jen po dobu nezbytně nutnou k vývoji nové funkcionality (feature).

1. Vývojář (tým) v rámci vývoje provede lokální změny v nové větvi.
   Vývojář (tým) změny otestuje buildem (sestavením) ve vlastním vývojovém prostředí.
   Vývojář (tým) změny potvrdí (commit).
2. V pravidelném intervalu se server CI dotazuje verzovacího serveru (tzv. polling), a tím se o změně dozví.
3. Server CI spustí všechny buildy (sestavení), které je potřeba provést při změně kódu.

4. Server CI spustí na nových buildech automatické testy.
5. Po doběhnutí automatických testů je na server CI publikován status testů včetně výsledků, které jsou všem přístupné.
6. Vývojář nebo tester řeší případné problémy v doběhnutí testů.

V případě bezchybného běhu a dokončení vývoje rozhodne vedoucí týmu o reintegraci feature větve do master větve (pepgotesting.com).



Obrázek 4 - Model technické implementace CI (pepgotesting.com)

3.2 Testování SW

Testování softwaru je obecně vnímáno jako proces hodnocení a porovnání softwaru s cílem odhalit rozdíly mezi daným vstupem a očekávaným výstupem. Tento proces by měl být samozřejmou součástí celého vývojového cyklu.

> *"A clever person solves a problem. A wise person avoids it."*
> *"Chytrý člověk problém řeší. Moudrý mu předchází."*
>
> Albert Einstein

Chápání testování se ale v různých zdrojích liší. Podle některých (Bourque, 2004) je testování dynamická kontrola, zda chování testovaného produktu odpovídá specifikaci. Dle této definice je k tomu, abychom daný produkt otestovali, nutné jeho spuštění, existence specifikace určující jeho chování a aplikace sady testovacích scénářů. Autoři (Bourque, 2004) zde mezi testování softwaru neřadí statické metody, ale považují je za nezbytný doplněk a dále je řadí mezi metody kontroly kvality softwaru.

Další zdroj (Hailpern, a další, 2001) přímo rozlišuje tři různé fáze: fázi ověřování, testování a validace.

Proces ověřování je zde definován jako proces dokázání a možné demonstrace, že produkt správně splňuje specifikace.

Vzhledem k tomu, že proces verifikace (ověřování) prokazuje shodu se specifikací, testování hledá případy, kdy produkt specifikaci neplní. Na základě této definice je možné nazývat testováním jakoukoli aktivitu, která vede k odhalení porušení specifikace produktu. V této souvislosti mohou být testováním nazývány i činnosti jako přehodnocení návrhu (design reviews), pravidelné kontroly kvality kódu a statické analýzy zdrojového kódu, přestože produkt zde není přímo spouštěn za účelem nálezu neočekávaných problémů. Vlastní spuštění kódu a testování jednotlivých testovacích scénářů je ovšem i zde označeno za hlavní součást testování.

Za validaci je pak považován proces hodnocení softwarového produktu na konci procesu vývoje s cílem ujistit se o souladu s požadavky (Hailpern, a další, 2001).

3.2.1 Kategorie testování

Testy softwaru lze rozdělit do několika kategorií. V těchto kategoriích se využívají různé kombinace druhů a typů testů. V následující části je uvedeno několik základních, často používaných kategorií testů. Uvedené rozdělení není jediné možné a často se lze setkat i s řadou dalších kategorií.

3.2.1.1 Statické a dynamické testování

Statickou analýzou zdrojového kódu bývá označen proces odhalování sémantických a logických chyb a získávání metrik kvality kódu. V kontextu kontinuální integrace jsou vybrané metriky získávány automatizovaně a je tak možné při využití vhodných nástrojů, sledovat jejich dlouhodobé trendy (Hujer, 2012).

Mezi často získávané metriky patří:

- počet řádků kódu, programových struktur
- složitost (komplexita) kódu
- dodržování standardů pro psaní kódu
- duplicitní kód.

V níže uvedeném citátu jeho autor, Bjarne Stroustrup, uvádí důvody, proč je důležité dohlížet na kvalitu kódu.

„I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well. “

Bjarne Stroustrup, autor jazyka C++ a knihy „The C++ Programming Language“

Zatímco statické metody analýzy kódu řeší zdrojový kód v jeho nezkompilované podobě, dynamické metody zkoumají již zkompilovaný spustitelný kód běžící v odpovídajícím prostředí a vyžadují tedy existenci spustitelné verze softwarového produktu. Vyhodnocování pak probíhá hlavně na základě poskytování různých vstupů a následného vyhodnocení výstupů (Borovcová, 2008).

3.2.1.2 Testování černé a bílé skříňky

Rozdělení testů na černou a bílou skříňku vychází z přístupu a ze znalostí s jakými tester přistupuje k testovanému produktu (Borovcová, 2008).

Testování již dokončených celků funkčního kódu bývá označováno jako testování černé skříňky (z anglického „black-box testing", známé také jako funkční testování). Černou skříňkou je zde produkt, do kterého nemá tester možnost pohlédnout a nemá k dispozici zdrojový kód. Smyslem těchto testů je ověření zadaných vstupů oproti očekávaným výstupům z pohledu uživatele bez hlubší znalosti vnitřní logiky produktu (Jenkins, 2008).

Do kategorie testu černé skříňky spadají téměř všechny druhy testů uživatelského rozhraní, akceptační testy a testovaní dle scénářů, které definují krok po kroku, jak by měl tester postupovat při testování určené funkcionality. V každém scénáři jsou jasně definované vstupy, uživatelem prováděné akce a očekávané výstupy. Výhodami testování typu černá skříňka jsou rychlost a snadnost, protože test může být prováděn i bez znalosti jakéhokoli programovacího jazyka. Nevýhodou pak může být riziko, že testovací scénář nepokryje všechny potenciální chyby (Pietrik, 2012).

Při testování bílé skříňky (z anglického „white-box testing", nebo známého jako strukturálním testování) nebo dle jiného názvu skleněné nebo průhledné skříňky (z anglického „glass box testing") je přístup odlišný. Testování spočívá v analýze samotného kódu - znalosti vnitřních datových a programových struktur, ale také vnitřní logiky softwaru (Pietrik, 2012). Testování typu bílá skříňka je často, ale ne vždy, v kompetenci programátorů. Používá techniky, které sahají od technicky specifického testování až po věci, jako je inspekce kódu (Borovcová, 2008).

Ačkoli techniky testování bílé skříňky mohou být použity v kterémkoliv stádiu životního cyklu softwarového produktu, obvykle se nacházejí v rámci jednotkových testů (Jenkins, 2008).

„Výhodou testování typu bílá skříňka je možnost odhalení nežádoucího kódu, a tedy vyšší kvalita kódu výsledného produktu." (Pietrik, 2012)

„Mezi těmito kategoriemi vznikla ještě třetí, testování šedé skříňky, kdy tester ví něco málo o implementaci a vnitřních pochodech produktu, ale ne tolik, aby to bylo považováno za

testování bílé skříňky. Například není k dispozici celý zdrojový kód, ale HTML kód webových stránek nebo informace o designu aplikace.“ (Borovcová, 2008)

3.2.1.3 Automatické a manuální testování

Další členění testování je možné na základě toho, zda jsou testy prováděny člověkem nebo softwarem, na automatické či manuální.

Při manuálním testování procházejí testeři softwarový produkt zcela náhodně nebo podle instrukcí předem daných testovacím scénářem a vyhodnocují, zda chování a výsledky produktu odpovídají očekávaným výstupům. Mezi výhody manuálního testování patří jeho jednoduchost a nenákladnost. Při dobré organizaci je možné testování i bez dalších investic do softwarových nástrojů.

„Manuální testování je pracný proces a od testera vyžaduje značnou dávku trpělivosti, všímavosti, kreativity a důmyslnosti. Díky lidskému důvtipu a nepředvídatelnosti lze v testované aplikaci docílit mnoha zajímavých stavů, které by se jinak obtížně navozovaly a otestovat tak chování aplikace v extrémních podmínkách.“ (Šolc, 2007)

I pro manuální testování lze využívat výhod plynoucích z držení se zásad kontinuální integrace. Každé úspěšné sestavení tak bude testerovi k dispozici pro manuální testy (Hujer, 2012).

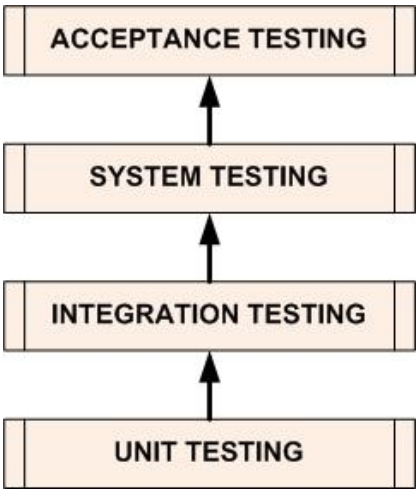
Naproti tomu princip automatického testování spočívá ve využití specializovaného nástroje pro vyvolání, zaznamenání a následné vyhodnocení požadované činnosti v testovaném produktu. Zásah testera je, v ideálním případě automatického testovaní, potřeba pouze ve fázi tvorby automatického testu, popř. při jeho vyhodnocování. Test samotný probíhá zcela nezávisle (Šolc, 2007).

Výhodou automatického testování mohou být díky automatizaci nižší náklady, menší časová náročnost a eliminace lidských chyb při testování.

3.2.1.4 Fáze testování

Na základě toho v jakém časovém odstupu od napsání kódu se testování provádí, lze testování rozdělit do těchto fází:

1. Jednotkové testování (Unit testing)
2. Integrační testování (Integration testing)
3. Systémové testování (System testing)
4. Akceptační testování (Acceptance testing)



Obrázek 5- Fáze testování

Jednotlivé stupně testování mohou být často prováděny rozdílnými lidmi a kód je zároveň na každé úrovni testován z odlišného úhlu pohledu.

První testování, které ovšem není formálně zahrnuté do uvedených fází, je vlastní testování kódu programátorem bezprostředně po jeho napsání. Při tomto testování programátor kontroluje, zda kód skutečně dělá, co zamýšlel a dále zkoumá, zda kód funguje správně, protože v této fázi je oprava vzniklých chyb nejméně nákladná.

V následujících fázích je kód již testován především z pohledu uživatele, tedy zda dělá to, co uživatel očekává a požaduje (Borovcová, 2008).

Jednotkové testování

Jednotkové testování (z anglického „unit testing—) je proces podrobného testování co nejmenších částí samostatně testovatelného kódu – tzv. jednotek. V procedurálním programování může být jednotkou funkce nebo procedura. U objektově orientovaného kódu se pak testují jednotlivé třídy a metody. Jednotkové testování nejčastěji provádí sám programátor, protože si tak může nejlépe a navíc okamžitě ověřit kvalitu vlastní práce (Pietrik, 2012).

Integrační testování

Integrační testování je proces ověřování správné interakce mezi novými i původními softwarovými komponentami.

„Integrační testování ověřuje, že nově přidané funkcionality spolu nekolidují a všechny jednotlivé podsystémy pracují správně, i když jsou zapojeny spolu s ostatními částmi systému.“ (Pietrik, 2012)

Systémové testování

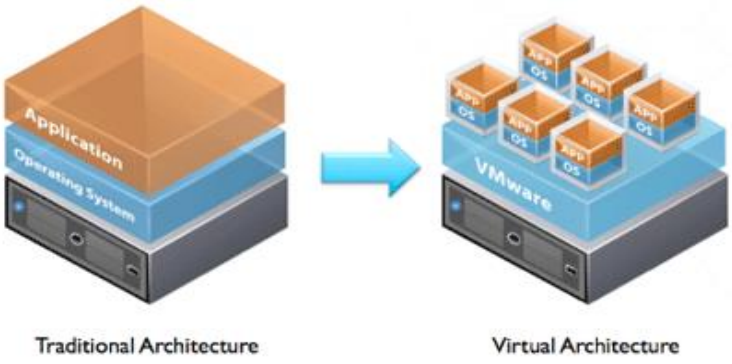
Systémové testování se zabývá chováním systému jako celku, proto bývají tyto testy prováděny spíše v pozdějších fázích vývoje. Většina problémů týkajících se funkcionality systému by již měla být identifikována předchozími fázemi testování. Zatímco jednotkové a integrační testování se zaměřuje především na testování funkcionality, při systémových testech je snaha simulovat pomocí rozsáhlých testovacích scénářů běžnou práci uživatele a tím otestovat systém i z hlediska bezpečnosti, přesnosti či spolehlivosti (Bourque, 2004). Do této skupiny testů patří mimo již zmíněných také testování výkonnosti, které slouží především k ověření, zda vyvíjená aplikace naplní či překročí aktuální či budoucí předpokládané požadavky uživatele a testování použitelnosti a přístupnosti (Pietrik, 2012).

Akceptační testování

Akceptační testování srovnává chování výsledného produktu s požadavky zadanými zákazníkem. Bývá prováděno v rámci převzetí produktu zákazníkem a slouží k ověření, zda je výsledný produkt připraven k nasazení v produkčním prostředí. Tato fáze testování může, ale také nemusí, zahrnovat práci vývojářů produktu. Obvykle se jím zabývá přímo zákazník nebo jím sestavený testovací tým (Pietrik, 2012).

3.3 Virtualizace

V předchozí části práce byl představen koncept kontinuální integrace a následně také úvod do testování SW, jakožto nedílné součásti. V další části si práce klade za cíl představit technologii virtualizace a osvětlit, proč je její využití užitečné v této oblasti.



Obrázek 6 - Srovnání tradiční architektury a virtualizace (VMware, Inc.)

V případě, že bychom chtěli definovat virtualizaci jako takovou, je možné říct, že je to vlastně oddělení zdrojů jako operační paměť nebo CPU od vlastních fyzických komponent, jak je vidět na Obrázek 6.

„Virtualization helps address your most pressing technology challenge: the infrastructure sprawl that's forcing IT departments to channel 70 percent of their budget into maintenance1 - and sapping resources for business-building innovation. The difficulty stems from the architecture of today's X86 computers: they're designed to run just one operating system and application at a time. That means that even small data centers have to deploy many servers - each operating at just 12 percent of capacity. That's highly inefficient by any standard. Virtualization software solves the problem by enabling multiple operating systems and applications to run on one physical server or "host." (VMware, Inc.)

Podle různých zdrojů se využití výpočetního výkonu dedikovaného serveru pohybuje mezi 6-12% (Gigaom). Virtualizace přináší možnosti jak mnohem efektivněji nakládat se zdroji. Výsledkem je pak úspora energie, snížení nákladů na provoz a flexibilnější management v celém IT odvětví, nejen v rámci vývoje SW. Nicméně právě výhodám, které virtualizace přináší do tohoto odvětví, se více věnuje tato práce.

Testování nových verzí aplikací je díky virtualizaci mnohem jednodušší. Každý vývojář si může na vlastním počítači vytvořit vlastní virtuální stanici a na ní nanečisto vyzkoušet běh aplikace. Virtualizace také usnadňuje testování multiplatformních aplikací nebo testování

více verzí současně ve stále čistém prostředí bez jakéhokoli rizika. V dřívější době bylo nutné pro tento účel vlastnit množství fyzických stanic či serverů, což dnes již není potřeba.

V neposlední řadě je nutné zmínit výhodu zálohování a monitoringu. V případě využití virtuálních disků lze celý systém kdykoli a kamkoli přenést nebo v případě potřeby vytvořit otisk aktuálního stavu a k němu se později vrátit v případě, že se některý krok nezdařil (Pavlis, a další, 2009).

3.3.1 Historie virtualizace

První pokusy s virtualizací probíhaly již od šedesátých let minulého století. V té době se běžně používal dávkový výpočetní model. Každý takovýto software byl svázaný s konkrétní hardwarovou a softwarovou konfigurací mainframů, na kterých byl spouštěn. V té době se ale v oblasti výpočetních prostředků děly neustále zásadní změny, takže každý nový hardware a k němu příslušný operační systém byl v principech odlišný od svého předchůdce. To mělo ovšem dopad na klienty, kteří byli nuceni s každou změnou hardwaru zároveň upravovat a měnit své dávkové aplikace. Právě kvůli široké škále hardwarových požadavků, začala firma IBM pracovat na systému S/360 – sálovém počítači, který by sloužil jako náhrada většiny ostatních systémů společnosti IBM a zároveň by byl navržen k zachování zpětné kompatibility. Dle prvního návrhu měl být tento systém jednouživatelský a určen k dávkovému spouštění úloh (Everything VM, 2011).

Odklon od dávkového zpracování začal 1. července 1963, kdy Massachusetts Institute of Technology (MIT) představil projekt MAC – Mathematics and Computation, později přejmenován na Multiple Acces Computer. Projekt MAC později získal významný grant od agentury amerického ministerstva obrany DARPA, který byl určen na financování výzkumu v oblastech operačních systémů, umělé inteligence a výpočetní teorie (Everything VM, 2011).

V rámci toho výzkumného projektu MIT potřeboval nový druh víceuživatelského počítačového hardwaru. Osloveni byli výrobci jako IBM, GE či Bell Labs, kteří v té době takové systémy nenabízely. IBM v té době spolupráci s MIT odmítla, protože společnost neviděla dostatečný potenciál a poptávku po takovém systému. MIT si proto vybral společnost GE, která správně odhadla potenciál takového systému.

Právě ztráta takové příležitosti byl hlavní impulz pro společnost IBM, která začala konečně vnímat poptávku po takovém produktu a to obzvlášť poté, co se ukázalo, že zájem o takový systém projevili Bell Labs. IBM se poté naopak zaměřila na vývoj technologií, kterým dnes říkáme virtualizace.

Prvním komerčně prodávaným systémem s podporou virtualizace byl systém IBM CP–67. Operační systém, který běžel na CP–67 se nazýval CP/CMS. CP, z anglického Control Program, byla hostitelská část, která vytvářela virtuální stroje, zatímco malý jednouživatelský CMS, z anglického Console Monitor Systém, byl navržen jako hostovaný virtuální stroj, který mohl komunikovat s uživateli. Interakce systému s uživatelem byla do té doby nevídaná. Před tímto systémem se IBM zaměřovalo na vývoj systému bez uživatelské interakce – programu se dodala data a ten vykonal operaci. Výsledek poté poslal do tisku nebo zobrazil na displeji. Oproti tomu interaktivní operační systém přidával uživateli možnost komunikovat s programy i za běhu.

Počáteční první verze CP/CMS byla vydána v roce 1968, první stabilní verze pak až v roce 1972 (Everything VM, 2011).

Sdílení prostředků v systému CP umožnilo každému uživateli mít svůj vlastní operační systém, díky čemuž se provádění operací a výpočtů mnohonásobně zjednodušilo. Toto řešení bylo v době jednouživatelských a jednoúlohových systémů opravdu revoluční. K dalšímu přelomu poté došlo až v osmdesátých letech s příchodem stolních počítačů a serverů řady x86, které již byly pro firmy a spotřebitele komerčně dostupné.

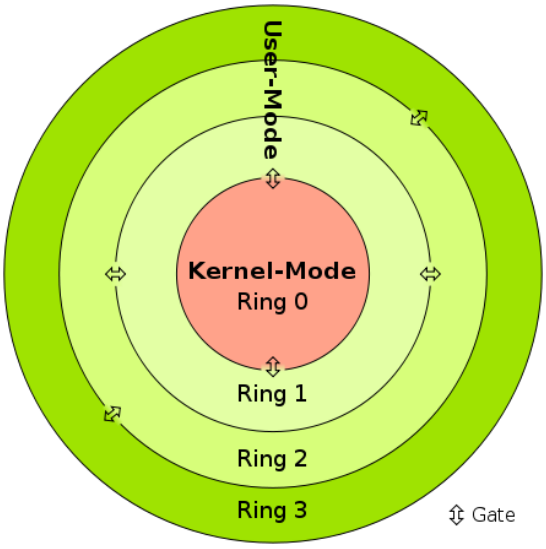
Růst trhu se stolními počítači vedl k poptávce po efektivnější a dostupnější virtualizaci, což stimulovalo vývojáře k dalšímu dynamickému technologickému rozvoji. V roce 1987 společnost Locus Computing Corporation vydává elementární virtualizační program Merge, který umožňuje spouštět MS-DOS v alternativních prostředích.

V devadesátých letech přispěla k postupnému rozvoji řada počítačových vývojářů a výrobců. K největším změnám ve vývoji této technologie ale došlo po roce 2000 a rovněž se také začal značně zvyšovat počet implementací. Dnes společnosti již běžně využívají nástroje pro virtualizaci hardwaru, desktopů, softwaru, pamětí, úložišť, dat a sítí.

3.3.2 Virtualizace na architektuře x86

Virtualizace systému na platformě x86 se z počátku potýkala s řadou problémů. Architektura procesoru (zde x86) hierarchicky rozděluje procesor do tzv. bezpečnostních úrovní (v angličtině protection domain neboli protection rings). Tento mechanismus má sloužit ke zvyšování odolnosti vůči chybám (fault tolerance) a zároveň k zabezpečení prováděných instrukcí.

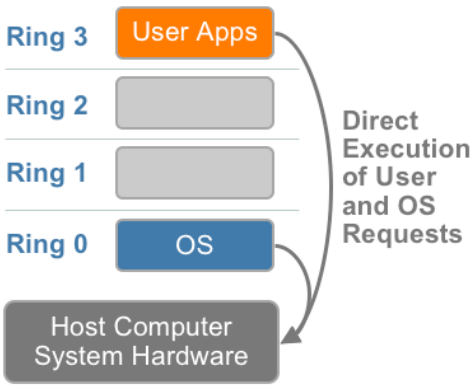
Hierarchie je rozdělena do základních 4 úrovní (ringů) – Ring 0 až Ring 3, jak je zobrazeno na Obrázek 7. Ve většině operačních systémů je právě úroveň 0 nejvíce privilegovaná s přímým přístupem k fyzickému hardware (Oracle). Takže zatímco v úrovni Ring 0 s právem spouštět privilegované instrukce, se předpokládá běh jádra operačního systému, v úrovni Ring 1 a Ring 2 běh jednotlivých ovladačů hardware, uživatelské aplikace běží až v neprivilegované úrovni Ring 3.



Obrázek 7 - Privilegované úrovně x86 architektury (Wikipedia)

Mezi jednotlivými úrovněmi jsou brány, které umožňují vždy pouze nejbližší úrovni přístup ke zdrojům a funkcím vrstvy sousední. Tato vrstvená architektura umožňuje zvýšit bezpečnost prováděných instrukcí. Pokud například běží škodlivý kód v uživatelské neprivilegované úrovni Ring 3, nemůže přímo používat služby nižších úrovní, jako

například síťové ovladače. Musí nejprve požádat sousední vrstvu o zprostředkování služby, což může omezit nebo zamezit jeho působení.

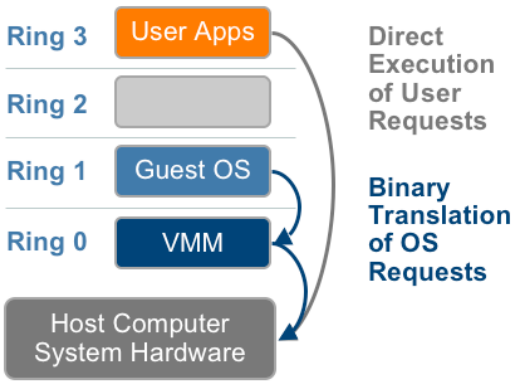


Obrázek 8 – Privilegované úrovně architektury x86 bez virtualizace (VMware, Inc., 2007)

Operační systémy na x86 platformě jsou navrženy tak, že vyžadují přímý přístup k hardwaru. Právě zde vzniká problém, protože při virtualizaci je potřeba umístit hypervisor do privilegované úrovně Ring 0, který může přímo přistupovat k hardwaru a dále ho přerozdělovat virtuálním strojům. Další problém nastal u instrukcí, které nebyly spouštěny z privilegovaného módu, takže nebyla možná jejich efektivní virtualizace. Zachytávání a překlad těchto instrukcí byl natolik náročný, že se zpočátku jevila virtualizace x86 architektury jako nemožná. Nakonec postupně vzniklo množství následujících přístupů, jak s těmito instrukcemi zacházet (Kábrt, 2012).

3.3.2.1 Plná virtualizace

Při plné virtualizaci jsou virtualizovány všechny součásti počítače. Veškeré požadavky operačního systému jsou tedy předávány hypervizoru a ten je za pomocí binárního překladu překládá a pouští na fyzickém hardwaru. Operační systém v tomto případě nemá možnost poznat, že je virtualizován. Hypervisor poskytuje virtuálním strojům všechny služby fyzického systému – virtuální BIOS, virtuální zařízení a správu pamětí. V případě plné virtualizace není třeba žádné modifikace operačního systému ani aplikačních programů.



Obrázek 9 - Privilegované úrovně x86 v případě plné virtualizace (VMware, Inc., 2007)

Dochází zde k plnému oddělení fyzické vrstvy, kdy veškeré programy běží pouze na virtuálním hardwaru a veškerý přístup k hardwaru je zprostředkován (Kábrt, 2012).

„To má samozřejmě řadu výhod - můžeme virtuální prostředí navrhnout tak, aby nám vyhovovalo (velikost paměti, typ procesoru, typ a kapacita disku apod.). Programy jsou rovněž nezávislé na konkrétním technickém vybavení, jeho změna nemá na virtuální prostředí vliv (samozřejmě kromě výkonnostních

charakteristik, tj. náš virtuální počítač může běžet rychleji nebo pomaleji, ale v každém případě poběží).“ (Matyska)

Tento druh virtualizace předpokládá podporu virtualizace na úrovni CPU.

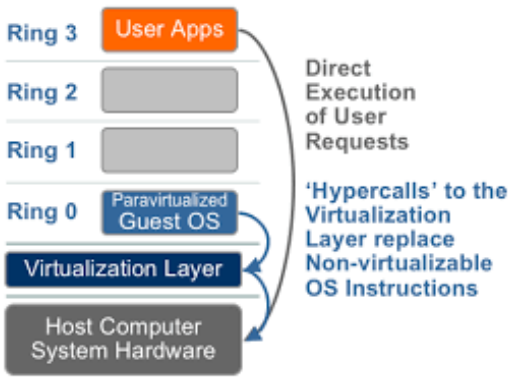
Podskupinou plné virtualizace je pak hardwarově asistovaná virtualizace, při které jsou speciální pomocné instrukce implementovány přímo do instrukční sady procesoru a nezatěžují tak hypervizor.

3.3.2.2 Paravirtualizace

"Para-" je přípona řeckého původu, které znamená "vedle", "s" nebo "po boku." Název paravirtualizace je tedy přímým odkazem na komunikaci mezi hostovaným operačním systémem a hypervizorem (VMware, Inc., 2007).

Paravirtualizace nepotřebuje ke svému běhu hardwarovou podporu virtualizačních technologií v CPU. Operační systém si je v tomto případě (na rozdíl od plné virtualizace) vědom, že je virtualizován. Modifikace jádra a to jak hostitelského, tak hostovaného operačního systému, tak aby byla možná úprava kritických operací, je zde proto nezbytná (Kábrt, 2012).

Modifikovaný operační systém využívající paravirtualizaci dosahuje většinou lepších výkonnostních výsledků v komunikaci s I/O zařízeními (síťová rozhraní, disková rozhraní atd.) než v případě plné virtualizace. Důvod je ten, že modifikovaný operační systém komunikuje v těchto případech přímo s ovladačem příslušného hardwarového zařízení přes hypervizora jako prostředníka pomocí tzv. hypercallů a ne přes emulovanou vrstvu jako při plné virtualizaci.



Obrázek 10 - Schéma fungování paravirtualizace na platformě x86 (VMware, Inc., 2007)

Produktem využívající této technologie virtualizace je např. XenServer, který bude představen dále v rámci praktické ukázky.

Nevýhody paravirtualizace vyplývají z nutnosti pozměněného jádra operačního systému, což není možné v případě operačních systémů firmy Microsoft, a ty v tomto případě nemohou být podporovány.

3.3.2.3 Virtualizace na úrovni operačního systému

V poslední době nabírá na oblíbenosti další z technik virtualizace, která z řady ostatních vybočuje a v mnohém se liší. Při virtualizaci na úrovni operačního systému nedochází k virtualizaci ani simulaci celého virtuálního stroje, ale jedná se v zásadě pouze o virtualizaci prostředí pro běžící aplikace. O separaci procesů a prostředků se zde stará jádro hostujícího operačního systému, které má přímý přístup k fyzickému hardware (Nový, 2007). Díky tomuto uspořádání je stejné jádro operačního systému využito pro implementaci hostovaných operačních systémů a jednotlivé virtuální stroje sdílí jádro hostovaného operačního systému i jeho služby. Aplikace, které pak běží v jednotlivých virtuálních serverech, jsou přesvědčeny, že běží v rámci samostatného systému. Režie této metody je prakticky neměřitelná (Bureš, 2009).
Mezi příklady této technologie patří např. OpenVZ a především čím dál více populární kontejnerová virtualizace Docker.

3.3.2.4 Hypervizor

Virtualizace umožňuje běh mnoha operačních systémů na jednom hostitelském počítači a právě k nekoliznímu paralelnímu běhu více operačních systémů slouží správce virtuálních operačních systémů, který se nazývá hypervizor. Tento hypervizor bývá označován také jako Virtual Machine Monitor (VMM) (Oracle, 2012). Hypervizor je tedy vlastně softwarová vrstva mezi fyzickým a virtuálním prostředím, která dále rozděluje hardwarové prostředky mezi hostované virtuální stanice (Tomášek, 2015).

Hypervizory jsou obvykle děleny na dva typy.

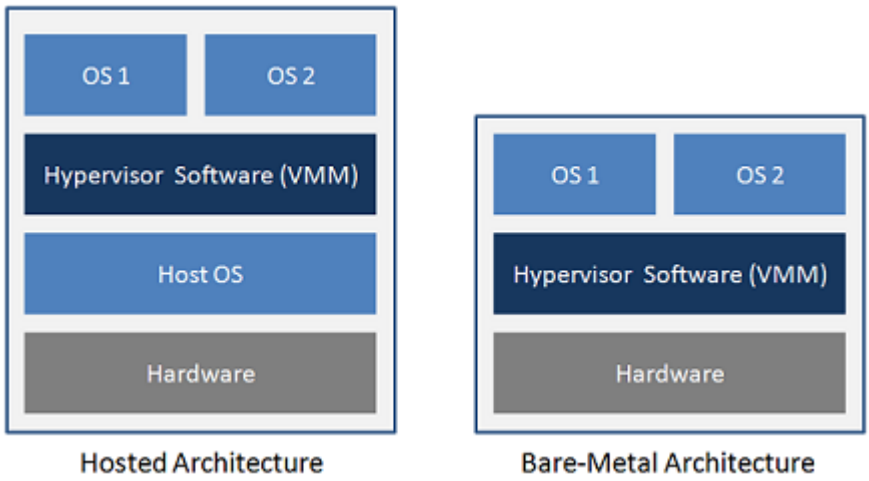
Nativní neboli bare-metal VMM

Prvním typem hypervizoru je tzv. nativní hypervizor, což je software, který běží přímo na hostitelském hardware a má jej pod kontrolou. Další jeho činností je monitorování hostovaných operačních systémů. Hostované operační systémy pak běží v samostatné vrstvě nad vrstvou hypervizora. Příkladem této implementace mohou být jmenovány např. produkty Oracle VM, Microsoft Hyper-V, VMWare, ESX a XEN (Oracle, 2012).

Hostovaný hypervizor

Druhým typem je hostovaný hypervizor. Ten je naopak postavený tak, aby běžel v další - třetí vrstvě nad tradičním operačním systémem. Využívá jeho „znalost“ hardwaru na kterém běží a sám vytváří pouze vrstvu mezi operačním systémem a hostovaným operačním systémem.

Nejznámějším příkladem hostovaného hypervizoru je např. Oracle VM VirtualBox nebo další jako VMWare Server, Microsoft Virtual PC, KVM, QEMU a Parallels.



Obrázek 11 – Hostovaný vs. Bare-Metal architektura (National Instruments, 2014)

3.4 Shrnutí poznatků

Předchozí části práce byly věnovány teoretickému úvodu vztahujícímu se k následné praktické části. Součástí představení vývoje softwaru bylo i rozebrání klíčových principů kontinuální integrace, kde jedním z těchto principů je i pravidelné testování nových buildů (sestavení). V návaznosti na toto testování je třeba mít tomu uzpůsobenou infrastrukturu. Nároky na infrastrukturu se mohou lišit v závislosti na vyvíjeném softwaru. Virtualizace pomáhá budovat infrastrukturu s vynaložením menšího úsilí a především financí než tomu bylo kdy dříve.

4 Vlastní řešení

V rámci vývoje webových aplikací je třeba zajistit pravidelné testování (viz. předchozí kapitola). K takovému testování je třeba vytvořit sadu skriptů, pomocí kterých bude možné nejprve vytvořit vhodné prostředí ke spuštění a následnému testování aplikace. Skripty by měly být zároveň lehce čitelné a spravovatelné pro každého člena týmu vývojářů.

4.1 Současné řešení

V současnosti používané řešení není vyhovující, protože neumožňuje spustit více testů zároveň a tím testovat ve více vývojových větvích současně. Navíc je tvořeno spletitou kombinací ANT skriptů, Groovy a bash skriptů a samotné zajištění infrastruktury trvá i hodinu.

Je zde také několik předpokladů, které je nutné dodržet.

Testovaná webová aplikace (není zde důležitý samotný účel aplikace) je pro účely testování získávána přímo ze serveru kontinuální integrace Jenkins v rámci pravidelných buildů. Aplikace je pro použití v rámci testování dodána ve formátu WAR (Web application Archive) a všechny zbylé potřebné soubory mohou být získávány z verzovacího systému SVN. Takto připravenou aplikaci je potřeba spustit v předem nakonfigurovaném prostředí. Toto prostředí by měl tvořit operační systém, a to konkrétně některá z distribucí operačního systému Linux, na něm běžící některý z podporovaných aplikačních serverů a v neposlední řadě zpřístupnění databáze, která bude k dispozici pro použití aplikace.

Řešení by se nemělo omezovat jen na určitý druh aplikačních serverů a databází, ale mělo by jich podporovat více, aby testování mohlo probíhat ve více kombinacích konfigurací (databáze 1 × aplikační server 1, databáze 1 × aplikační server 2 atp.). Požadovanou vlastností celého řešení je tedy možnost dynamické volby vlastní konfigurace, ve které bude aplikace testována.

Aplikace může být provozována jako tzv. stand-alone aplikace či v režimu cluster o n-uzlech, čemuž je třeba přizpůsobit možnosti konfigurace.

Na začátku samotného procesu je třeba získat pro konkrétní konfiguraci několik parametrů. Dle hodnot těchto parametrů je možné dále rozhodovat, jaký bude další postup.

Název parametru	Možné hodnoty	Popis
TEST_ENV_TYPE	server | cluster	Nastavení typu nastavení aplikace.
NODES_COUNT	1…n	Počet uzlu aplikace.
CONTAINER	Tomcat8, Jetty9	Typ webového kontejneru.
DATABASE	MySQL, PostgreSQL, derby...	Typ použité databáze.
CREATE_NEW_VM	true | false	Nutnost vytvoření nového VM.
EXISTING_VM_NAME	< jméno VM >	
RUN_TEST	true | false	

Tabulka 1 - Základní parametry před během testu

Fungování celého procesu vytvoření infrastruktury pro potřebné testy je možné zobrazit diagramem na Obrázek 12. Diagram mnoho kroků ještě nezahrnuje a to především v části nastavování jednotlivých VM[1]. Tyto kroky budou podrobněji rozebrány v průběhu řešení. Očekává se také, že proměnných bude nakonec do systému vstupovat mnohem více.

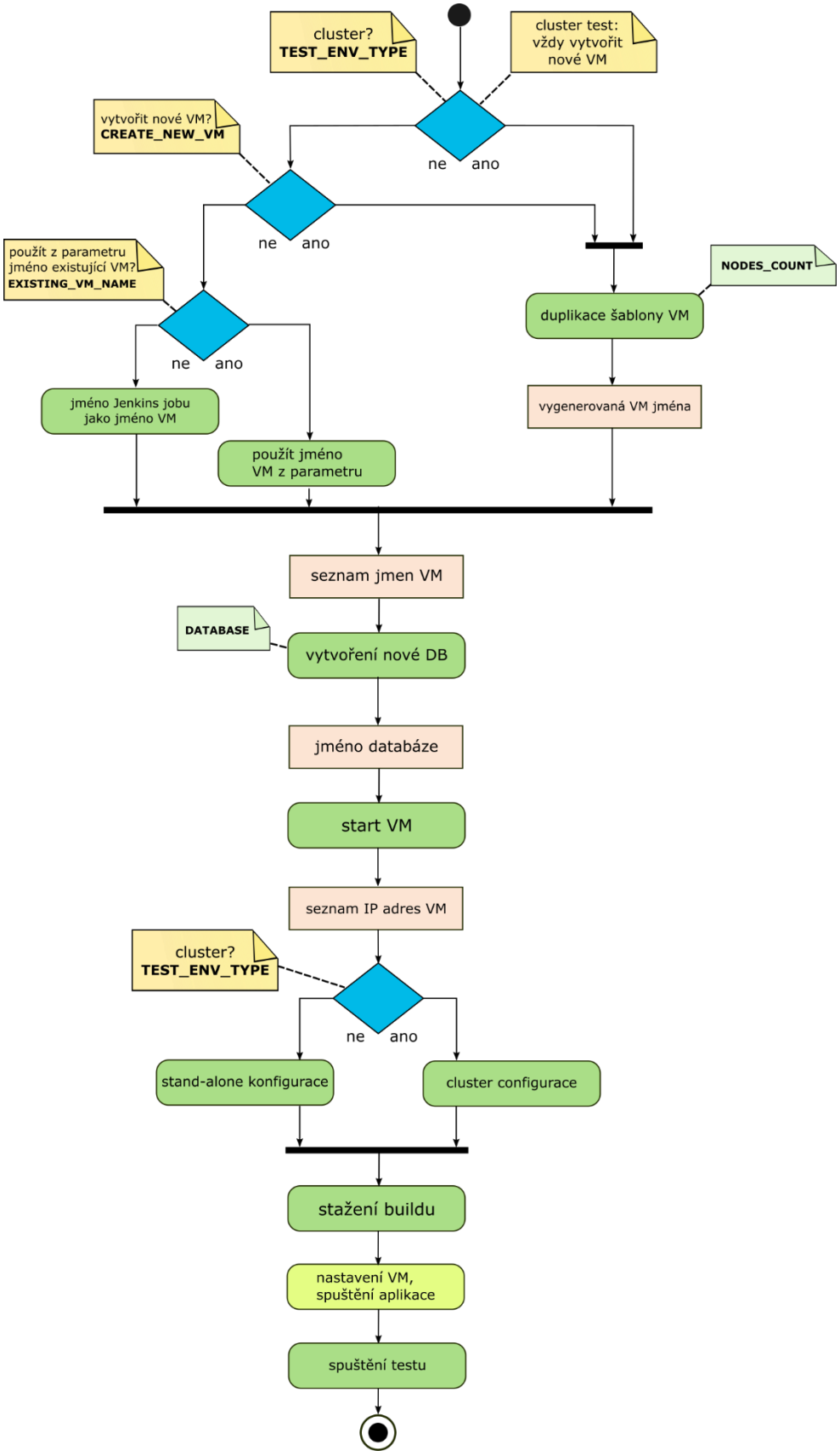
V následujících částech práce proběhne seznámení s některými specifiky jednotlivých součástí, kterým by měla být věnována pozornost – např. specifika práce s CI Jenkins, XenServer či aplikačními servery.

Praktické řešení bude ilustrováno ve dvou variantách.

Některé kroky řešení jsou specifické pouze pro testovanou aplikaci. Tyto kroky budou zobecněny nebo alespoň řádně označeny.

Obě varianty budou počítat s využitím co nejméně nástrojů třetích stran z důvodu přenositelnosti a jednoduššího znovu zprovoznění např. na jiném serveru.

[1] Virtual Machine



Obrázek 12 - Diagram procesu vytvoření infrastruktury od startu ke spuštění testu

4.2 Seznámení s prostředím

V rámci prostředí je již dána virtualizační platforma – tou zde bude řešení firmy Citrix XenServer. Server s hypervizorem Xen je již v existujícím prostředí zprovozněn a používán i pro další virtuální stanice.

4.2.1 Citrix XenServer

Společnost Citrix zaujímá v současnosti vedoucí postavení na trhu produktů, které poskytují všechny důležité funkce a nástroje pro implementaci virtualizace serverů.

Zásadní průlom ve vývoji společnosti se odehrál v roce 2007, kdy byla Citrixem realizována akvizice společnosti XenSource, lídrem v oblasti řešení podnikové virtuální infrastruktury s hlavním produktem Xen Project (Citrix).

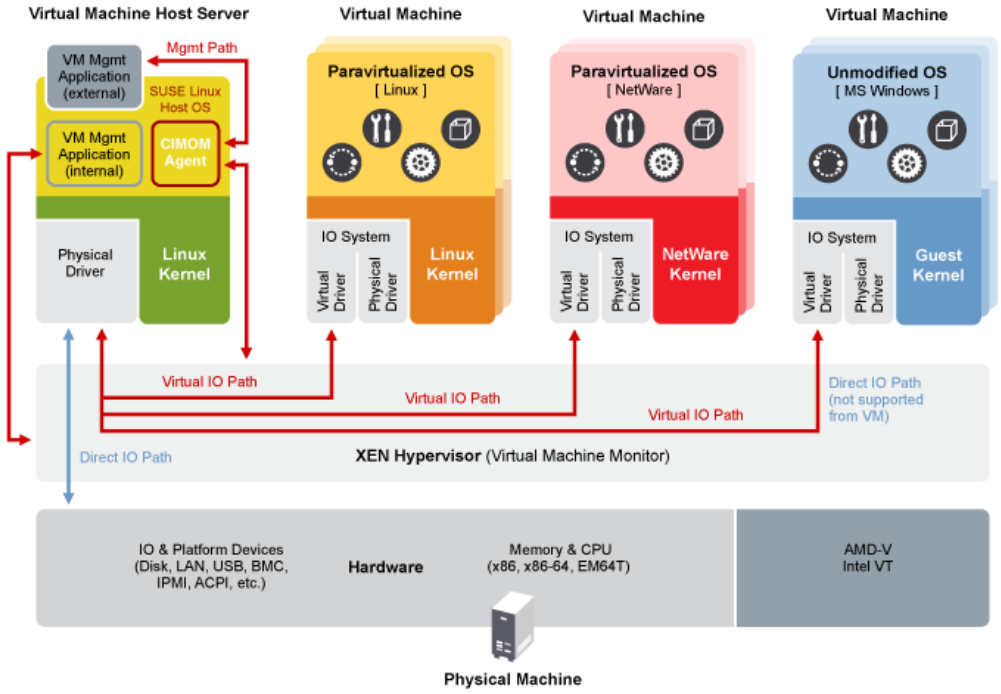
Hypervisor Xen Project™ je bare-metal virtualizační platforma využívána produktem XenServer široce uznávaným jako nejrychlejší a nejbezpečnější podniková virtualizační platforma.

Oproti open-source Xen Project™ hypervizoru Citrix XenServer obsahuje některé další funkce jako např. (Citrix, 2015) :

- Control domain – privilegovaný virtuální systém Dom0 sloužící pro běh souboru nástrojů - *xapi*
- XenCenter – client pro management virtuálních strojů pro operační systémy
- šablony s předkonfigurovanými parametry pro optimální běh nejoblíbenějších operačních systémů
- Storage Manager – vestavěná podpora pro velké množství úložných zařízení
- podpora kontejnerové virtualizace Docker.

Jak již bylo uvedeno výše, Citrix XenServer je založen na open-source projektu Xen Hypervisor a umožňuje efektivní provoz a správu Windows a Linuxových virtuálních strojů. XenServer běží přímo na serveru bez požadavku na operační systém, čímž vzniká efektivní a škálovatelný systém (Citrix, 2015).

Princip XenServer je dobře ilustrován Obrázek 13 znázorňujícím architekturu systému. XenServer sám kontroluje hardwarové prostředky a dle potřeby je dále přiděluje virtuálním stanicím.



Obrázek 13 – XenServer – architektura systému (Citrix)

4.2.1.1 Administrace XenServeru

Správa XenServeru je možná pomocí uživatelsky přívětivějšího grafického klienta XenCenter pro operační systémy Windows nebo pomocí XenServer CLI (Command Line Interface) známé jako „xe“ pro operační systémy Linux.

Tyto nástroje umožňují např.:

- správu i monitoring všech XenServer serverů, sloužících jako hostitelské systémy
- kompletní správu virtuálních strojů od jejich vytváření, klonování, monitoring až po jejich zánik
- sledování výkonnostních statistik celého systému
- live-migrace virtuálních stanice mezi jednotlivými hostitelskými servery.

XenServer CLI se jeví jako ideální právě při automatizaci úkolů jako je klonování virtuálních stanic apod. v rámci řešeného úkolu. XenCenter je na druhou stranu ideální pro použití při základním monitoringu.

XenServer CLI

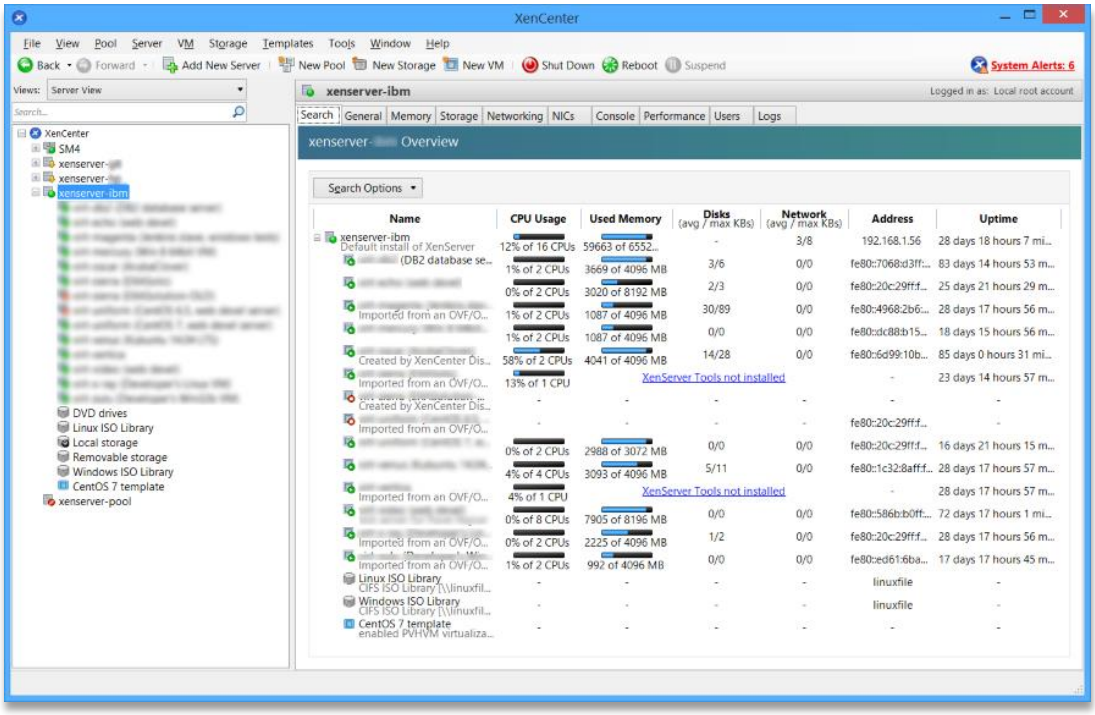
Možnosti příkazů rodiny xe jsou rozsáhlé. Kompletní seznam příkazů lze získat přímo na manuálových stránkách XenServeru, ale existují také tzv. „Command Reference“. Jednotlivé příkazy jsou logicky rozděleny do skupin dle typu ovládaného prvku:

- příkazy *pool* k ovládání skupin hostujících serverů XenServer
- příkazy *host* k ovládání jednotlivých hostujících serverů Xenserver
- příkazy *vm, template, snapshot* k práci s jednotlivými virtuálními stroji v rámci jednoho serveru XenServer
- skupina příkazů *sr, vdi, vdb, pbd, sm* pro ovládání úložných zařízení
- skupina síťových příkazů jako *network, bond, vlan, tunnel* a další

Konkrétní práce s těmito příkazy bude demonstrována dále při vlastním řešení úkolu.

XenCenter

XenCenter je jako klient pro operační systémy Windows určen pro přímý monitoring správcům systémů. XenCenter obsahuje kompletní přehled o využití zdrojů, ale i statistiky výkonu nebo možnost přímého přístupu k jednotlivým virtuálním stanicím přes konzoli. Zatímco využití XenCenter při automatizaci úkolů je téměř nemožné, jako nástroj pro správu je ideální.



Obrázek 14 - Klient pro OS Windows pro správu hostitelských serverů XenServer

4.2.2 Jenkins CI

Tomu, proč je kontinuální integrace ve vývoji softwaru důležitá, byla v práci věnována celá kapitola již dříve. I v této části byl zmíněn zřejmě nejpoužívanější server kontinuální integrace Jenkins CI. Tato část si klade za cíl představit již lehce konkrétněji práci s Jenkins CI.

Server je napsaný v Javě a původně byl určen rovněž pro kontinuální integraci projektů napsaných v Javě. Obsahuje tedy velmi silnou podporu nástrojů určených pro projekty napsané právě v Javě, nicméně nic nebrání využití i pro jiné projekty.

S růstem popularity (jak kontinuální integrace, tak Jenkins CI) vzrostl a stále roste i počet pluginů, kterými je možné systém rozšiřovat a přizpůsobovat vlastnímu použití. Množství pluginů vyvíjené aktivní komunitou vývojářů je jednou z hlavních předností Jenkins CI. Těchto pluginů existují stovky a plugin lze najít téměř na cokoli, na co si člověk vzpomene, včetně takových pluginů jako je Lava Lamp Notifier plugin (Jenkins, 2016) na zobrazení statusu buildu pomocí lávové lampy připojené do počítače přes USB port.

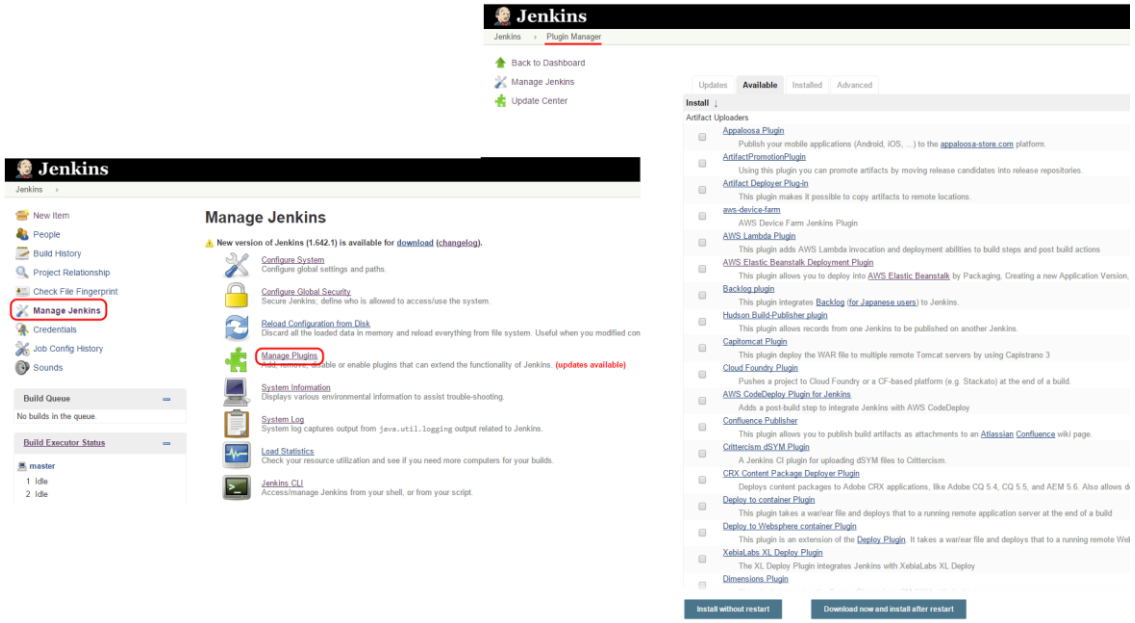
Projekt Jenkins CI je primárně určen pro menší projekty o několika vývojářích, ale lze jej bez problémů použít i na obrovských projektech, kde jsou stovky a tisíce jobů. Příkladem může být např. veřejný server Jenkins společnosti Apache[2] (AbcLinuxu, 2011).

Bylo by zde také na místě uvést některé důležité pojmy, které budou dále zmiňovány:

- *Job* – v rámci Jenkins CI může existovat bezpočet jednotlivých úloh, které jsou v anglické mutaci Jenkins označovány jako Job
- *Build* – v rámci každého jobu je možné spouštět buildy - tedy jednotlivá sestavení
- *Build step* – každý build může sestávat z více kroků; tyto kroky mohou být provedeny v předem definovaném pořadí
- *Artifact* – v rámci buildu vznikají soubory – v anglické mutaci Jenkins „build artifacts“ – tedy soubory vygenerované sestavením
- *Workspace* – pracovní složka každého jobu

[2] http://builds.apache.org

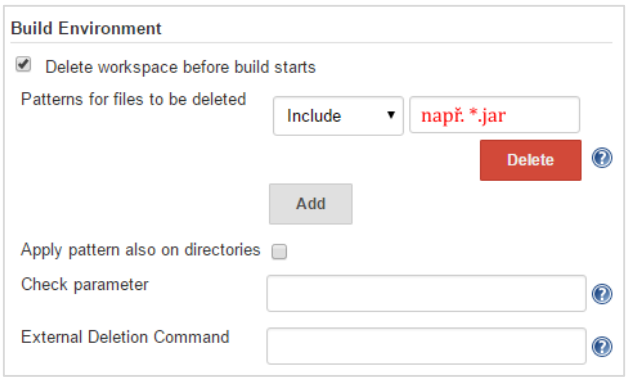
4.2.2.1 Instalace a konfigurace



Obrázek 15 - Instalace plug-inů do Jenkins CI

Instalaci Jenkins CI zde nebude věnováno mnoho prostoru, protože v rámci prostředí řešené problematiky Jenkins CI server již nainstalovaný byl a je běžně používán.

Základní instalace je ovšem velice jednoduchá. Na stránkách projektu (Jenkins, 2016) jsou přímo ke stažení nativní balíčky pro jednotlivé operační systémy či distribuce systému. Po instalaci je ovšem Jenkins CI poměrně holý a veškeré další funkcionality je nutné doinstalovat do systému pomocí tzv. plug-inů. Tyto pluginy je možné opět najít přímo na webových stránkách projektu či je možné je nainstalovat přímo z prostředí Jenkins CI jak lze vidět na Obrázek 15.

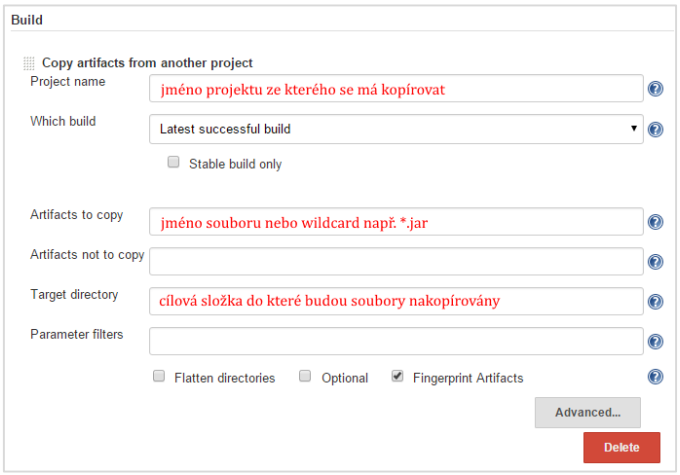


Obrázek 16 - Parametry konfigurace Workspace Cleanup pluginu

Z užitečných dodatečných pluginů lze jmenovat např. **Workspace Cleanup Plugin**, pomocí kterého je možné vyčistit workspace před každým novým spuštěním. Tím lze zamezit některým nepředvídatelným chybám způsobených neaktuálností souborů.

Copy Artifacts plugin umožňuje v rámci běhu buildu zkopírování artefaktů z jiného jobu včetně možnosti volby konkrétních artefaktů a konkrétní cílové destinace.

Další, pro účely práce nezbytný plugin, je **Subversion Plugin**, který umožňuje stahovat soubory z verzovacího systému SVN. Nastavení SVN pluginu v rámci jobu je již jednoduché a stačí pouze zadat URL SVN repositáře, ze kterého má Jenkins stahovat soubory a cestu kam je má ukládat.



Obrázek 17- Konfigurace Copy Artifact Pluginu v rámci jobu

Potřeba dalších pluginů může vzniknout až při samotném řešení a nemá smysl jich zde více uvádět.

4.3 Postup řešení

V rámci diplomové práce budou představeny dvě varianty možného řešení. Díky porovnání variant by měly být zřejmé výhody či nevýhody odlišných přístupů. Obě varianty se budou lišit především v použití odlišného způsobu virtualizace.

V první variantě bude hrát významnou roli vlastní vytvořená šablona se samostatným operačním systémem a již přednastavenými aplikačními kontejnery, která bude dle potřeby klonována a do které bude aplikace nahrána pomocí doprovodných skriptů vygenerovaného nastavení a spuštěna.

Druhá varianta bude využívat virtualizace na úrovni operačního systému Docker, která se liší především v tom, že předem není nakonfigurována žádná virtuální stanice ani šablona, ale existuje pouze jakýsi předpis pro vytvoření kontejneru. V takto vytvořeném kontejneru bude poté připraven i aplikační nebo webový server a aplikace je následně nahrána. I toto se děje pomocí sady doprovodných skriptů.

Přestože je u obou variant počítáno s jistou univerzálností, co se týče aplikačních serverů, některé z těchto serverů mají svá specifika. Rozsah této práce, ale nedovoluje věnovat se všem těmto podrobnostem, a proto bude pro účely této práce postup demonstrován pouze

na dvou z nich vybraných. Konkrétně na Apache Tomcat 8 a Jetty 9. Přestože Apache Tomcat není uváděn jako aplikační server, pro potřeby této práce je aplikačním serverem cokoliv, co implementuje Java EE specifikaci nebo její část a proto jím je zde brán i Apache Tomcat.

Podobně široká podpora je očekávána také u množství typů databází, se kterými má být aplikace testována. Pro účely práce není důležitá vlastní konfigurace databází, a proto zde bude předpoklad, že všechny dané databáze jsou již nakonfigurovány se stejnými přihlašovacími údaji.

Některá nastavení, jako např. konfigurace databázového spojení, předávána pomocí konfiguračních souborů aplikaci, mohou být specifická pouze pro konkrétní testovanou aplikaci. Protože je to součástí celého procesu, tato nastavení bude práce řešit také.

Pro všechny pomocné skripty v obou variantách bylo zvoleno použití příkazového jazyku Bash. Bash je obecně velice dobře čitelný, a proto splňuje jeden z hlavních předpokladů – jednoduchou čitelnost a udržovatelnost.

V rámci řešení nebude řešena vlastní tvorba nebo spouštění testů. Práce si za cíl dává pouze tvorbu infrastruktury s možností následně spouštět libovolné testy např. voláním dalšího skriptu.

4.4 Varianta 1 – předpřipravená šablona VM

4.4.1 Příprava šablony VM

XenServer již v základu obsahuje řadu předpřipravených šablon pro různé operační systémy. Není tedy příliš potřeba se zabývat samotnou instalací operačního systému Linux. Nehraje zde roli ani výběr distribuce, protože skripty by neměly obsahovat nic specifického pouze pro některou distribuci.

Pro účely práce byla přímo v grafickém klientovi XenCenter vytvořena virtuální stanice s operačním systémem Ubuntu LTS 14.04. V rámci nastavení této virtuální stanice je nutné určit parametry, jako počet CPU, množství paměti, či velikost diskového oddílu.

Tyto velikosti mohou být později kdykoli měněny.

Instalovaná virtuální stanice bude instalována bez jakéhokoli grafického prostředí, protože to pro tento účel není potřeba.

Oba aplikační servery jsou nainstalovány pouhým stažením a extrahováním do složky */opt/* viz Tabulka 2. Následně jsou v nově vytvořené složce */opt/test* vytvořeny symlinky se zkráceným názvem pro zjednodušení upgradů verze aplikačních kontejnerů. V budoucnu nebude při přechodu na vyšší minor verzi potřeba změny všech skriptů, ale pouze aktualizace symlinku.

Aplikační server	Plná cesta k jeho instalaci	Vytvořený symlink
Jetty 9	/opt/jetty-distribution-9.3.7.v20160115	/opt/test/jetty9
Tomcat 8	/opt/apache-tomcat-8.0.24	/opt/test/tomcat8

Tabulka 2 - Systémové cesty k aplikačním kontejnerům v rámci virtuální stanice

Protože jsou oba aplikační servery ve výchozím nastavení nastaveny, aby aplikace běžela na portu 8080, je třeba upravit konfigurační soubory.

U Jetty 9 je třeba upravit soubor **start.ini** nacházející se přímo v kořenové složce Jetty a konkrétně odkomentováním části:

```
## Connector port to listen on
jetty.http.port=8081
```

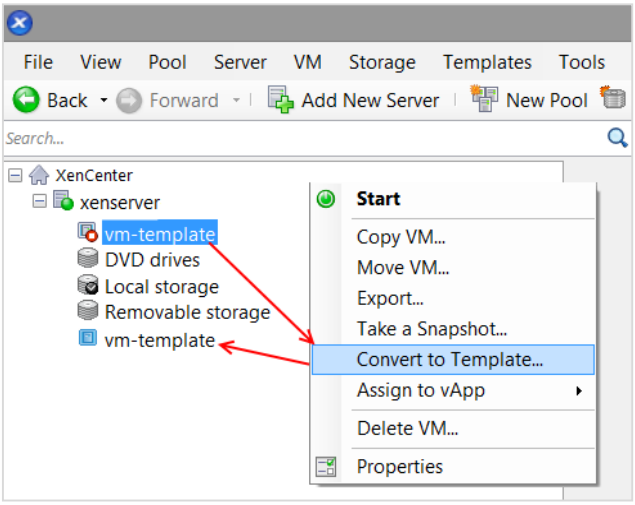
Tímto bude aplikace spuštěná v Jetty 9 dostupná na portu 8081.

V případě Apache Tomcat 8 je třeba upravit soubor **server.xml** ve složce **conf**.

```
<Connector port="8082" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

Tímto nastavením bude aplikace spuštěná v Apache Tomcat 8 dostupná na portu 8082.

Veškerá ostatní nastavení mohou být ponechána ve výchozím stavu.



Obrázek 18 - Vytvoření šablony VM z existujícího VM

V tomto stavu je již možné z existujícího a nastaveného virtuálního stroje udělat šablonu.

Vytvoření šablony v XenCenter je pak již jen otázka pár kliknutí. Virtuální stroj je nejprve nutné vypnout pomocí akce *Shutdown VM* z kontextového menu nebo přímo z konzole VM. Poté se objeví v kontextovém menu možnost *Convert to Template*, čímž se virtuální stanice změní na šablonu, viz Obrázek 18.

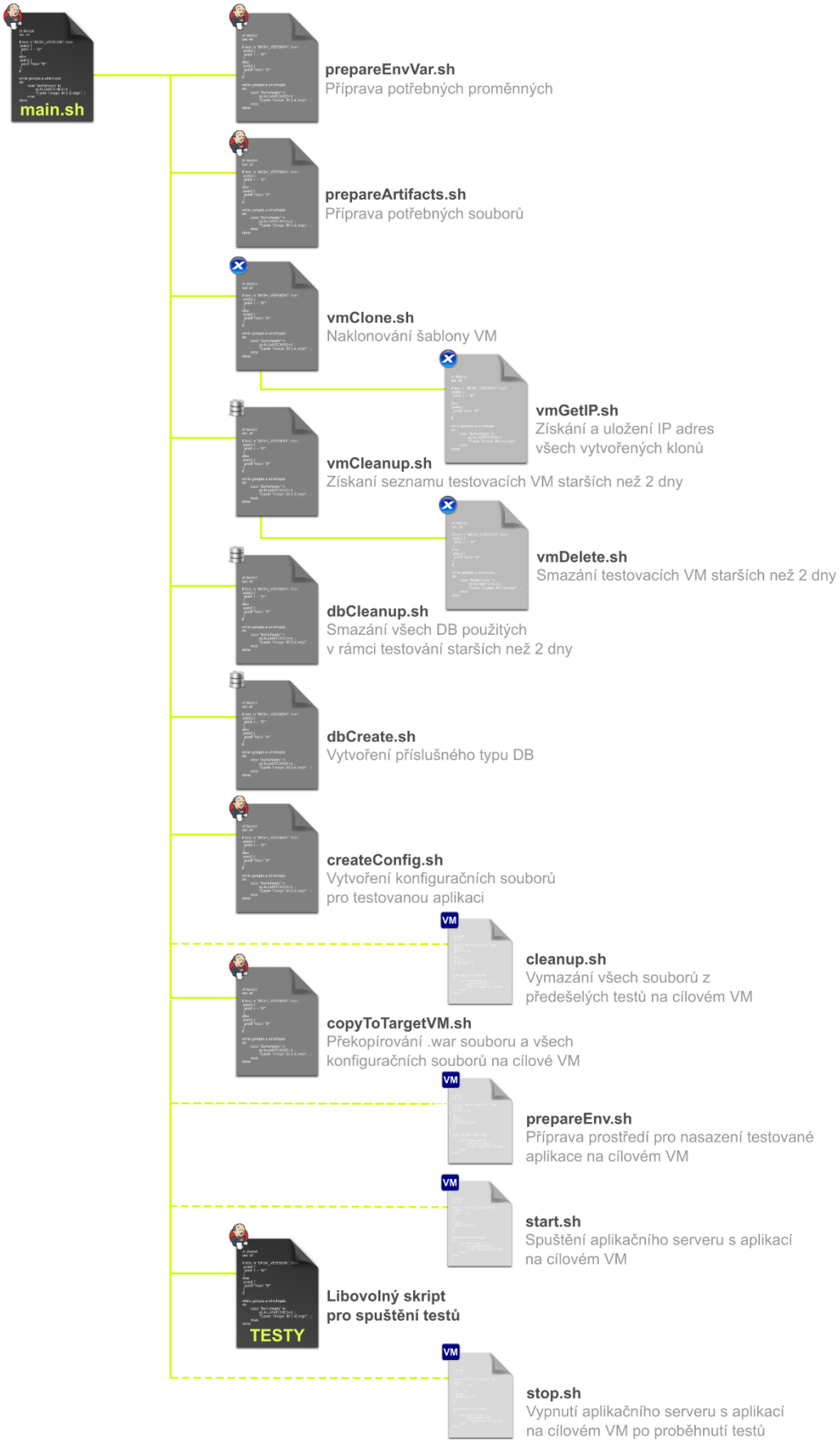
4.4.2 Řešení dílčích problémů

Celé řešení se skládá ze souboru skriptů, které jsou pouštěny z hlavního skriptu **main.sh**. Přestože by celé řešení mohl obsáhnout jeden skript, členění do více skriptů dle řešených částí přispívá k větší přehlednosti.

Při celém řešení bylo dbáno i na možnost snadného dohledání problému a každý skript obsahuje na svém začátku výpis svého názvu, názvu stanice, na které skript běží a některé další informace. Stejný výpis se objeví i na konci skriptu a je tak jednoznačně možné identifikovat skript, ve kterém se případný problém vyskytl.

V rámci procesu jsou skripty postupně pouštěny ve více prostředích. Některé skripty jsou pouštěny přímo na serveru Jenkins CI, některé jsou kopírovány na XenServer pro další akce, některé pracují také s databázovým serverem a některé skripty jsou spouštěné až na výsledném klonu původní šablony. Z Obrázek 19 by mělo být patrné jak pořadí spouštění skriptů, tak i právě prostředí ve kterém je skript spouštěn.

V rámci této práce nebude věnována pozornost zabezpečení možnosti připojení se na tyto stanice/ servery a rovnou zde bude předpoklad, že je toto připojení vždy realizovatelné pomocí SSH připojení.



Obrázek 19- Ilustrace postupného spouštění jednotlivých skriptů

4.4.2.1 Konfigurace Jenkins jobu a skript main.sh

Hned na začátku skriptu je potřeba definice potřebných proměnných.

Některé proměnné předává při spouštění jobu přímo Jenkins CI. Tyto proměnné jako je číslo buildu (BUILD_NUMBER), název jobu (JOB_NAME) nebo URL Jenkins CI (HUDSON_URL) budou využity dále především pro identifikaci jednotlivých naklonovaných VM.

Mimo tyto proměnné je potřeba Jenkins jobu **test-app-job-v1** nadefinovat dalších sedm parametrů. Parametry umožňující výběr mezi několika možnostmi Choice Parameters, String Parameter umožňující do proměnné předat libovolný textový řetězec a Boolean Parameter předávající hodnotu true nebo false.

Z parametrizovaného jobu v Jenkins CI jsou pak předány parametry určující, zda se bude jednat o standalone konfiguraci či clusterovou (TEST_ENV_TYPE), v případě clusterové pak i počet uzlů (NODES_COUNT), zvolený aplikační server (CONTAINER), typ databáze (DATABASE) a parametry určující zda se použije již existující VM (EXISTING_VM_NAME) nebo bude vytvořen nový klon (CREATE_NEW_VM). Nastavení Jenkins jobu s těmito parametry je vidět na Obrázek 20.

Mimo tyto proměnné je potřeba definovat cesty k souborům, které jsou pro běh jobu stažené z verzovacího systému. Tyto proměnné budou používány v rámci celého procesu.

```
#JENKINS PATHS
CONFIG=$WORKDIR/config.properties       # the final configuration file made of
other parts
SCRIPTDIR=$WORKSPACE/testenv            # directory with scripts from SVN
PROPERTIES=$SCRIPTDIR/property-files    # folder with files with DB and others -
these comes from
CONFIG_FILES=$SCRIPTDIR/config-files    # pre-prepared config files
CONT_SCRIPTS=$SCRIPTDIR/property-files/container-scripts   # folder with files
with DB, CONTAINER settings
#TARGET SERVERS PATHS
CONTAINERHOME=/opt/test/$CONTAINER      # container directory on target server
DEPLOYDIR=/tmp/deploy/$CONTAINER        # directory where all files are copied
from Jenkins
```

Obrázek 20 - Nastavení parametrizovaného Jenkins jobu

4.4.2.2 Příprava potřebných proměnných

V tomto kroku je ze skriptu main.sh spuštěn skript **prepareEnvVar.sh**, který připraví další potřebné proměnné.

Jednou z těchto potřebných proměnných je jméno databáze, která bude později vytvořena a použita pro konfiguraci aplikace. Kvůli limitaci délky názvu pro jeden z požadovaných typů databáze – DB2, musí jméno databáze obsahovat maximálně 8 znaků (IBM).

Celé jméno je, pro částečnou unikátnost i identifikovatelnost, složeno ze dvou částí – z posledních čtyř znaků ze jména Jenkins jobu s vynecháním pomlček a nahrazením všech velkých písmen za malé – opět kvůli limitacím názvu DB2 databáze.

```
JOB_NAME_DB=`echo $JOB_NAME | tr -d "-" | rev | cut -c1-4 | tr
'[:upper:]' '[:lower:]' | rev`
```

Druhá část názvu je náhodná čtyřmístná sekvence čísel vygenerovaná pomocí interní funkce bashe RANDOM, která vrácí pseudonáhodnou číselnou hodnotu v rozmezí 0 - 32767 (tldp.org) – v tomto specifickém případě je definované rozmezí od 0 do 9999.

```
#the final name
export DBNAME="$JOB_NAME_DB$((RANDOM%9999))"
```

Další proměnné, které je potřeba získat, jsou proměnné specifické pro jednotlivé aplikační servery. Protože takových proměnných může být v budoucnu více než jen momentální proměnná PORT specifikující port, na kterém aplikace běží, budou všechny tyto proměnné sdružovány v jednom souboru **containers.properties**. Každá proměnná bude mít prefix určující, k jakému aplikačnímu serveru patří. Viz příklad níže.

```
#tomcat8
tomcat8.PORT=8082

#jetty9
jetty9.PORT=8081
```

Proměnné už jen pro vybraný aplikační server jsou překopírovány do dočasného souboru. Pomocí příkazu *source* jsou pak tyto proměnné nastaveny.

```
grep "$CONTAINER\." $CONFIG_FILES/containers.properties | grep -v '^#' |
cut -d. -f2- > $WORKSPACE/tmp.properties && source
$WORKSPACE/tmp.properties
```

Posledním krokem v tomto souboru je rozhodnutí, zda se jedná o clusterovou konfiguraci. V takovém případě, se budou vždy generovat nové klony VM a proměnná CREATE_NEW_VM je nastavena na true. Pokud se jedná o standalone konfiguraci, je proměnná určující počet uzlů NODES_COUNT nastavena na 1.

```
if [ "$TEST_ENV_TYPE" == "cluster" ];then
      export CREATE_NEW_VM=true
      else
            export NODES_COUNT=1
fi
```

4.4.2.3 Příprava potřebných souborů

Po dokončení předchozího skriptu je následně zavolán skript **prepareArtifacts.sh**, který má za úkol získat všechny potřebné artefakty z Jenkins CI nebo z workspace.

Pro stažení artefaktů z Jenkins CI obsahuje jednoduchou funkci *downloadFile*.

```
#function to download and extract each of defined files
function downloadFile () {
RESULT=""
ATTEMPT=0
until [ "$RESULT" == "Success" ]; do
  if [[ $ATTEMPT -lt 3 ]];then
      DIR=$WORKDIR/$2
      mkdir -p $DIR
      URL=$HUDSON_URL/job/$1/lastSuccessfulBuild/artifact/$2/$3
      echo -e  "Downloading $URL to $DIR/$2"
      wget -q $URL -O $DIR/$3 && RESULT="Success" || RESULT="Failure"
      echo $RESULT
      ATTEMPT=$(( ATTEMPT + 1 ))
      else
            echo -e "\n**** Could not download arttifacts"
            exit 1
  fi
done
}

#the list of files to download for tested application
downloadFile "test-app-job" "dist" "test.war"
```

Tato funkce očekává tři parametry – název jobu a název složky, ze které budou stahovány artefakty a název stahovaného souboru. Aby bylo zamezeno předčasnému chybovému ukončení skriptu z důvodu jakéhokoli dočasného síťového výpadku, funkce se snaží soubor získat z Jenkins třikrát. Funkci je možné použít pro stažení libovolného počtu artefaktů z různých jobů.

V rámci tohoto skriptu je také připraven properties soubor s parametry, které budou využity později.

4.4.2.4 Naklonování šablony VM

Poté co jsou všechny artefakty, včetně souboru s parametry, připraveny je předchozí skript skončen. V rámci main.sh jsou pak skripty překopírovány přímo na XenServer, kde je spuštěn skript **vmClone.sh**

Nejprve je zde po řádcích v rámci cyklu while přečten v předchozích krocích připravený soubor *variables.properties*.

```
while read LINE; do
  declare -x $LINE 1>/dev/null
done < $XENDIR/variables.properties;
```

Poté je již možné přikročit k samotnému naklonování šablony VM.

V prvním kroku je třeba rozhodnout, zda se jedná o clusterové prostředí nebo standalone.

Pokud se jedná o clusterové prostředí, je třeba vytvořit tolik klonů, kolik je požadováno clusterových uzlů.

```
count=$NODES_COUNT;
type="cluster-node"
genName=`</dev/urandom tr -dc A-Za-z0-9 | head -c4`

#duplicate the VM n-times with generated names
    for (( i = 1 ; i <= count ; i++ ));
        do
            num_formatted=`printf "%02d" ${i}`
            VMName=$BRANCH-$genName-$type$num_formatted
            echo -e "\n**** Creating $VMName"
            xe vm-install new-name-label=$VMName template=vm-template;
            echo -e "\n**** Starting $VMName..."
            xe vm-start vm=$VMName;
            #print VMName
            VMlist[$i]=$VMName
        done
            VMlistGenerated=`echo "$(printf '%s,' "${VMlist[@]}")"`
            VMlist=$VMlistGenerated
```

Seznam vygenerovaných jmen každého z n ($NODES_COUNT) uzlů je uloženo do proměnné **VMlist**. Tento seznam je později použit pro doplnění IP adres.

V tomto kroku již dochází k faktickému naklonování šablony a nastartování VM. Kopie VM je vytvořena pomocí příkazu *xe vm-install* s parametry jména klonu a názvu šablony a následně je nové VM nastartováno příkazem *xe vm-start*.

Situace je o něco složitější, jedná-li se o standalone konfiguraci kde je generován pouze jediný klon. Z diagramu na Obrázek 12 vyplývá, že je nutné rozhodnout, zda je potřeba vytvořit nový klon (CREATE_NEW_VM). Pokud tento parameter není nastaven, je třeba také vyhodnotit, jestli by testy neměly běžet na již existujícím VM (EXISTING_VM_NAME). V případě, že tento parametr není definován, bude použit VM se stejným jménem jako Jenkins job, který test spustil. Pokud definován je, bude test puštěn na VM nastaveného jména. Pokud takový VM neexistuje, bude vytvořen. Vytvoření nového VM probíhá pomocí funkce newVM a opět s použitím příkazů *xe vm-install* a *xe vm-start*. I zde je výstupem název VM v proměnné **VMlist**.

```
function newVM () {
        VMName=$1
        echo -e "\n**** Creating $VMName"
        xe vm-install new-name-label=$VMName template=vm-template;
        echo -e "\n**** Starting $VMName..."
        xe vm-start vm=$VMName;
        VMlistGenerated=$VMName
}
#TEST_ENV_TYPE = server
if [ "$CREATE_NEW_VM" == "true" ]; then
#New VM with generated name
genName=`</dev/urandom tr -dc A-Za-z0-9 | head -c4`
newVM test$genName-$BUILD_NUMBER
VMlist=$VMlistGenerated
      else
#CREATE_NEW_VM=false
      if [ -n "$EXISTING_VM_NAME" ]; then
      echo -e "\n**** Using EXISTING_VM_NAME=$EXISTING_VM_NAME as name
      for VM"
      VMlist=$EXISTING_VM_NAME
      else
            echo -e "\n**** Using JOB_NAME=$JOB_NAME as name for VM"
            VMlist=$JOB_NAME
      fi
# check if desired VM really exists, if not, create it first
XenVMList=`xe vif-list params=vm-name-label | cut -d: -f2 | grep
"${VMlist}" | xargs | head -n1`
      if [ "$XenVMList" == "$VMlist" ]; then
      xe vm-start vm=$VMlist;
      else
            echo "The VM with the specified name does not exists and
            will be created."
            newVM $VMlist
      fi
fi
```

Seznam IP adres klonů

Poté co jsou všechny virtuální stanice vytvořeny a nastartovány, je potřeba získat jejich IP adresy pomocí skriptu **vmGetIP.sh**.

Protože jedním z požadavků byla minimalizace externích nástrojů jako např. k podobnému účelu stvořený *fping* nebo příkaz *nmap*, řešením nakonec bylo použití standardních příkazů *ping* a *arp*. Příkaz ping nejprve pošle na každou IP adresu v definované podsíti dotaz a čeká na odezvu protistrany. Způsob použití pomocí funkce multiping, umožňuje mnohem rychlejší dokončení všech dotazů.

```
#ping all addresses in subnet
function multiping()
    {
        ping -c1 192.168.0.$COUNT 2>&1 >/dev/null
    }

#ping all IP addresses in subnet in parallel to update arp table
pingSubnet () {
echo -e "\n**** Pinging addresses to update arp table"
    COUNT=1
    while [ $COUNT -lt 255 ] ;
    do
        multiping &
        let COUNT=COUNT+1
    done
wait
}

#call the function pingSubnet
pingSubnet
```

Tímto je aktualizována ARP tabulka. Protokol ARP slouží v rodině protokolů TCP/IP k získání linkové adresy síťového rozhraní protistrany (MAC adresy) ve stejné podsíti pomocí známé IP adresy.

Pomocí příkazu *xe vif-list*, který primárně slouží k získání informací o VM, je získána MAC adresa pro již známé VM a poté je z ARP tabulky pomocí příkazu *arp* vypsán této MAC adrese odpovídající záznam a získána tak i IP adresa. Protože aktualizace ARP tabulky může někdy selhat, je v rámci funkce počítáno s více pokusy.

```
for VM in $VMlist
    do
    echo -e "**** Getting MAC and IP address for VM $VM"
    MAC=`xe vif-list vm-name-label=$VM params=MAC device=0 |tail -c 20`
    IP=`arp -n | grep -i "$MAC" | cut -f1 -de`
    ATTEMPT=1
```

```
    while [ ! -n "$IP" ];
      do
        if [ $ATTEMPT -le 5 ];then
        echo -e "**** Attempt #${ATTEMPT} to get IP of the VM $VM."
        pingSubnet
        let ATTEMPT=ATTEMPT+1
            else
            echo -e "\n**** IP address for VM $VM cannot be obtained."
            exit 1
        fi
      IP=`arp -n | grep -i "$MAC" | cut -f1 -de`
      done
      #create complete list with VMname,IP
      VMlistGenerated[$i]=`echo "$VM,$IP"`
      IPlistGenerated[$i]=`echo -e "$IP" | cut -d: -f1-4`
      echo -e "**** IP address for VM $VM is $IP"
      i=$i+1
done
```

Po získání všech adres jsou IP adresy uloženy do proměnné IPlist a činnost skriptu tím končí.

```
printf -v IPlist "%s\n" "${IPlistGenerated[@]}"
```

Po dokončení **vmGetIP.sh** se v rámci skriptu **vmClone.sh** zapíše obsah proměnné IPlist do souboru ip.list a ten se zkopíruje zpět do workspace Jenkins jobu.

V tomto bodě jsou již vytvořeny všechny virtuální stroje a jsou k nim k dispozici jejich jména a IP adresy. Zpět ve skriptu **main.sh** je třeba ověřit, zda opravdu existují soubory se jmény a IP adresami.

4.4.2.5 Seznam VM starších než 2 dny

Před dalšími kroky je třeba myslet také na automatickou údržbu infrastruktury. Při opakovaném běhu testů by neustále přibývaly naklonované virtuální stanice i vytvořené databáze. Tomu je třeba zabránit a zajistit automatický úklid.

Pro účely testování by nemělo být potřeba vytvořené prostředí dlouho udržovat. Výsledky testů by měly být zkontrolovány bezprostředně po doběhnutí jobu a případné akce, ke kterým by mohla být použita databáze nebo VM, by měly být dokončeny také co nejdříve. Životnost VM a databáze bude tedy 2 dny a poté budou smazány.

K účelu mazání VM slouží skript **vmCleanup.sh**. Na databázovém serveru (v současném prostředí nejlépe dostupný a příhodný) byl vytvořen adresář testVM, kde existuje seznam virtuálních stanic a data jejich vytvoření. Jakmile je virtuální stanice vytvořena v rámci

tohoto skriptu, je zkontrolováno, zda již v seznamu **allVM.list** existuje. Pokud ano, řádek se přepíše záznamem s novým datem, pokud ne, vloží se nový.

```
if grep -Fq "$VM_NAME" allVM.list; then
sed -i "s|.*$VM_NAME.*|$VM_NAME,$CREATED|g" allVM.list
else
    echo "$VM_NAME, $CREATED" >> /testVM/allVM.list
fi
```

Poté je seznam procházen po řádcích a pomocí krátké funkce v jazyce ruby je vypočítáno, zda je rozdíl mezi datem vytvoření a aktuálním datem větší než dva dny.

```
#check all from allVM.list and delete those which are older than 2 days
while read line;
do
    #check how old VMs are
    CREATIONDATE=`echo $line | cut -d, -f2`
    VM=`echo $line | cut -d, -f1`
    TODAY=`date +"%y-%m-%d"`
    DIFF=$(ruby -rdate -e "puts Date.parse('$TODAY') -
Date.parse('$CREATIONDATE')")
    DAYDIFF=`echo $DIFF | cut -d/ -f1`

      if [ "$DAYDIFF" -gt "2" ]; then
      echo -e "**** VM $VM has been created on $CREATIONDATE and is
$DAYDIFF days old and therefore will be removed."
      ssh root@xenserver "cd $XENDIR; ./vmDelete.sh $VM;" < /dev/null
      sed -i /$VM/d /testVM/allVM.list
      fi
done < allVM.list
```

Pokud je VM starší než dva dny, je spuštěn skript **vmDelete.sh** s parametrem jména virtuálního stroje a ten je pomocí příkazu *xe vm-shutdown* nejprve vypnut a poté smazán pomocí příkazu *xe vm-uninstall*. Oba příkazy potřebují jako hlavní parametr identifikační číslo VM UUID, které lze získat pomocí příkazu *xe vm-list*.

```
UUID=`xe vm-list is-control-domain=false is-a-snapshot=false | cut -d: -
f2 | grep "$VM" -B1 | xargs | cut -d" " -f1`
xe vm-shutdown uuid=$UUID;
sleep 10s
xe vm-uninstall uuid=$UUID force=true;
```

Po smazání virtuálního stroje je smazán i řádek v souboru allVM.list.

Jakmile se přečte celý seznam allVM.list, skript vmCleanup.sh je dokončen a pokračuje main.sh.

4.4.2.6 Smazání DB starších než 2 dny

Obdobně je řešeno i pravidelné mazání databází starších dvou dnů. Na rozdíl od mazání VM existuje v případě databází seznam pro každý typ databáze (DATABASE) zvlášť.

Ten je opět po řádcích pročítán a obdobným způsobem je získána informace, zda je databáze starší než 2 dny. Pokud je databáze starší, je pomocí case konstrukce zavolán příkaz k jejímu smazání. *Case* podobně jako *if* umožňuje vytvořit konstrukci, ve které se budou provádět za různých okolností různé příkazy nebo skupiny příkazů.

```
export DBLIST=/testDB/$DATABASE.list
while read line;
do
#check how old is database
      CREATIONDATE=`echo $line | cut -d, -f2`
      export DBNAME=`echo $line | cut -d, -f1`
      TODAY=`date +"%y-%m-%d"`
      export DIFF=$(ruby -rdate -e "puts Date.parse('$TODAY') -
      Date.parse('$CREATIONDATE')")
      export DAYDIFF=`echo $DIFF | cut -d/ -f1`

  if [ "$DAYDIFF" -gt "2" ]; then
      echo -e "\n**** The database $DBNAME has been created on
      $CREATIONDATE and is $DAYDIFF days old and will be removed"
      #commands to delete DB
  case "$DATABASE" in
   'mysql')
     mysql -utest --execute "DROP DATABASE $DBNAME;";RESULT=$?;;
   'postgresql')
     dropdb -Upostgres $DBNAME;RESULT=$?;;
   'db2')
     if [ -f ~/.profile ]; then . ~/.profile;fi;$DB2_HOME DROP DATABASE
     ${DBNAME};RESULT=$?;;
   'derby')
     rm -rf /opt/derby/$DBNAME;RESULT=$?;;
     esac
  fi
done < $DATABASE.list
```

Po smazání databáze je smazán také řádek v seznamu a tím je činnost skriptu **dbCleanup.sh** ukončena.

```
sed -i /$DBNAME/d  $DATABASE.list
```

4.4.2.7 Vytvoření DB pro testovanou aplikaci

Předcházejícím krokem je ukončena činnost na XenServeru a dočasně vytvořená složka s potřebnými skripty je smazána, aby ani zde nezůstávaly neaktuální a již nepotřebné soubory.

Dalším krokem je vytvoření nové databáze pro aplikaci k čemuž slouží skript **dbCreate.sh**.

Skript se pomocí SSH připojuje na server, kde jsou nainstalovány všechny potřebné databázové systémy a pomocí konstrukce case je spuštěn příkaz dle typu databáze (DATABASE) k jejímu vytvoření. Tyto příkazy (i příkazy k mazání z předešlého skriptu) lze najít v manuálových stránkách příslušné databáze.

```
case "$DATABASE" in
 'mysql')
   ssh user@db-server "mysql -utest --execute 'CREATE DATABASE IF NOT
   EXISTS ${DBNAME};'" < /dev/null ;;
 'postgresql')
   ssh user@db-server "createdb -Upostgres ${DBNAME}" < /dev/null;;
 'db2')
   ssh user@db-server "export DB2INSTANCE=db2inst; $DB2DIR/bin/db2
   CREATE DATABASE ${DBNAME} PAGESIZE 32768" < /dev/null;;
esac
```

Po vytvoření databáze je přidán řádek s názvem databáze a dnem vytvoření do příslušného seznamu pro pozdější mazání viz předchozí krok.

```
CREATED=`date +"%y-%m-%d"`
ssh user@db-server "echo "$DBNAME, $CREATED" >> /testDB/$DATABASE.list"
< /dev/null
```

4.4.2.8 Použití získaných parametrů v konfiguraci aplikace

Skript pro vytvoření konfigurace se spouští v rámci jednoho běhu tolikrát, kolik bylo na začátku požadováno (v případě clusterové konfigurace) uzlů (NODES_COUNT). Protože je jedním z následně použitých potřebných proměnných také IP adresa aktuálního VM, cyklem while se po řádcích prochází seznam IP adres všech vytvořených klonů a pro každý aktuálně vybraný se tak vytvoří jeho vlastní konfigurace.

Tento krok je velice specifický pro testovanou aplikaci a proto bude pro účely této práce počítáno s tím, že by obecně aplikace potřebovaly některé z údajů jako např. vlastní IP adresu (IP), port (PORT), domovský adresář aplikačního server (HOME), v jednom z předchozích kroků vygenerovaný název již vytvořené databáze (DBNAME). V případě clusterové aplikace zde bude počítáno s předpokladem, že by každý z uzlů potřeboval vědět IP adresu jakéhosi „master“ uzlu (IPFIRST), což je první vytvořené VM v seznamu IP adres.

K tvorbě výsledného konfiguračního souboru jsou tak použity jejich šablony s placeholdery na místech, kam se pomocí jednoduché funkce doplní aktuální informace.

```
CONFIG=$WORKDIR/config.properties

function replaceString
{
    echo "**** Replacing *$1* by *$2* in the file $CONFIG"
    sed -i -e "s|$1|$2|g" $CONFIG
}
```

Soubor *config-standalone.properties* je určen pro údaje jako domovský adresář, IP adresa atp. pro typickou instalaci, *config-cluster.properties* obsahuje údaje pro spuštění v cluster módu a poslední soubor *db*.properties je soubor obsahující informace pro připojení k databázi.

V případě, že se jedná o cluster instalaci, jsou do jednoho výsledného souboru *config.properties* spojeny všechny tři jmenované soubory, jinak jsou spojeny pouze db.properties a config-standalone.properties.

```
if  [ "$TEST_ENV_TYPE" == "cluster" ]; then
cat $CONFIG_FILES/config-standalone.properties db.properties
$CONFIG_FILES/config-cluster.properties > $CONFIG
    else
        cat $CONFIG_FILES/config-standalone.properties db.properties >
$CONFIG
fi
```

Následně jsou v nich pomocí funkce *replaceString* nahrazeny placeholdery skutečnými hodnotami.

```
##CLUSTER CONFIGURATION
replaceString CLUSTER_MASTER_IP=IPFIRST CLUSTER_MASTER_IP=$IPFIRST

##INITIAL CONFIGURATION
replaceString HOME $HOME
replaceString IP $IP
replaceString PORT $PORT

##DB CONFIGURATION
replaceString DBNAME $DBNAME
```

Na konci skriptu je připraven soubor *config.properties* se všemi potřebnými údaji.

4.4.2.9 Překopírování potřebných souborů na cílovou VM

Dalším krokem ,v rámci zmíněném while cyklu, je překopírování všech do tohoto kroku připravených artefaktů na cílový virtuální stroj. Nejprve je nutné cílové VM připravit. Pro

ten účel jsou na klon nakopírovány pomocné skripty, které se liší pro každý aplikační server.

cleanup.sh

Jakmile jsou skripty nakopírovány na VM, je zde spuštěn skript **cleanup.sh**. Ten slouží k vyčištění prostředí tak, aby zde nezůstaly již žádné soubory po případném předchozím běhu. Prvním krokem je také zastavení aplikačního serveru, pokud běží, dále je nutné smazat všechny předchozí konfigurační soubory, verze aplikace nebo logy.

V dalším kroku je spuštěn samotný skript **copyToTargetVM.sh**, který pouze pomocí programu *scp* překopíruje všechny artefakty na cílový virtuální stroj. Artefakty nejsou kopírovány přímo do domovského adresáře aplikačního serveru, ale pouze do dočasné složky.

prepareEnv.sh

Poté je spuštěn již přímo v složce aplikačního serveru skript **prepareEnv.sh**, který připravené soubory překopíruje do cílové destinace v adresáři aplikačního serveru.

```
DIR=/tmp/deploy/jetty9
HOME=/opt/testas/jetty9
HOTDEPLOYDIR=$HOME/webapps

# move war to deploy dir
cp -v $DIR/test.war $HOTDEPLOYDIR/test.war

# config file to container's home folder
cp -v $DIR/config.properties $HOME/config.properties

if [[ -f $HOTDEPLOYDIR/test.war && -f $HOME/config.properties ]];then
echo "**** test.war and config.properties are OK!"
    else
        echo "**** test.war and config.properties are missing!"
fi
```

V tomto okamžiku je tak cílová virtuální stanice kompletně připravena a je možné přejít k dalšímu uzlu nebo v případě samostatné instalace tento krok ukončit a přejít k dalšímu kroku.

4.4.2.10 Pomocné skripty spouštěné na cílové VM

start.sh

V tomto okamžiku jsou již nastaveny a připraveny všechny cílové VM a je potřeba pouze nastartovat aplikační servery a tím aplikaci na každém z nich nasadit.

Opět k tomu poslouží cyklem while po řádcích čtený seznam IP adres.

Ten v prvním kroku vždy spustí přímo na cílovém VM skript **start.sh** a poté pomocí dalšího jednoduchého while cyklu testuje každých 10s odezvu aplikace a čeká na její kompletní spuštění (HTTP status kód 200). V případě, že aplikace nenastartuje do 300s, není možné dál pokračovat a celý job je neúspěšně ukončen.

```
export SERVERURL=http://$IP:$PORT/test
APPSTATUS=0
COUNTER=0

echo -e "\n**** Waiting for test startup on $SERVERURL."
while [ "$APPSTATUS" != "200" ];
    do
      sleep 10s
        COUNTER=`expr $COUNTER + 1`
        TIME=`echo "$COUNTER * 10" | bc`
        echo "$TIME seconds..."
            if [ $COUNTER == "30" ];then
            echo -e "\n**** Something went wrong, the server is not
            running on $SERVERURL , PID: $$ PPID: $PPID , $HOSTNAME "
            echo "Server status=$APPSTATUS"
            exit 1
            else
               if test -z "$APPSTATUS"
               then
                  APPSTATUS=0
               fi
            fi
APPSTATUS=` curl -s -I $SERVERURL | grep HTTP/1.1 | awk {'print $2'}``
done
```

V tomto momentě, pokud všechny kroky proběhly řádně, by měly již na všech potřebných naklonovaných VM běžet řádně nastavené aplikační servery s nasazenou aplikací test.war, která má přístup do databáze. Nyní je možné spustit vůči této aplikaci jakékoli testy, což může být např. puštěním nějakého dalšího skriptu apod. Tento skript není součástí této práce a může být libovolný.

stop.sh

Poté co doběhnou i všechny testy, je možné všechny aplikační servery opět vypnout spuštěním skriptu **stop.sh** na každém VM ze seznamu IP adres.

4.4.3 Zhodnocení varianty

Při použití varianty se šablonou virtuální stanice došlo k značnému zrychlení. Celkově příprava virtuální stanice, konfiguračních souborů a poté nasazení cílové aplikace zabere do pěti minut.

Stinnou stránkou této varianty je potřebná údržba šablony. V případě potřeby jakékoli změny v konfiguraci nebo např. přidání dalšího aplikačního serveru, je potřeba virtuální stanici sloužící jako předloha přenastavit, odladit a znovu z ní vytvořit šablonu. Tento postup je zdlouhavý a náchylný na chyby.

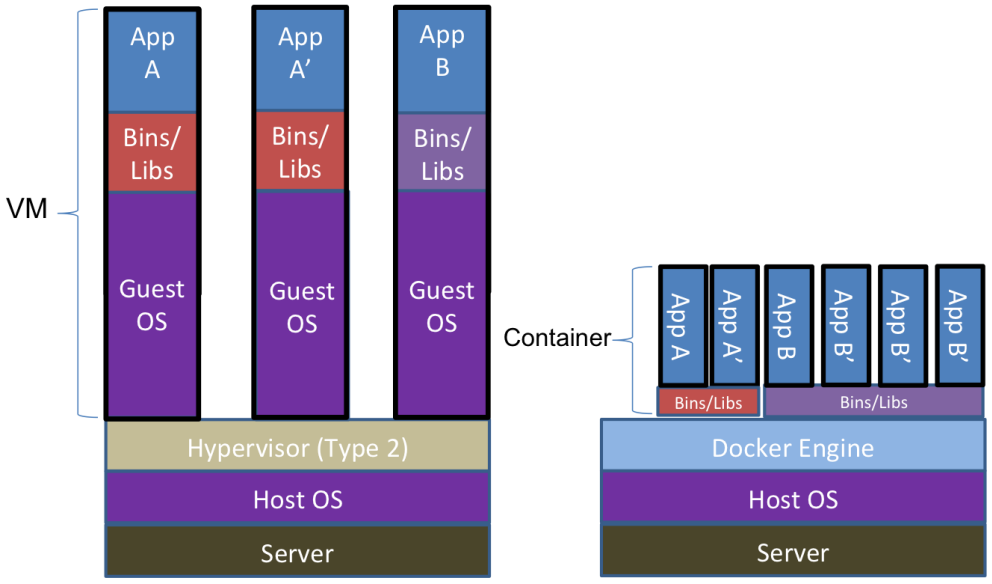
V současné době se proto od podobných řešení upouští a cílem je dosáhnout bodu, kdy je veškerá infrastruktura tzv. v kódu. Tento přístup vede nejen k větší flexibilitě při změnách, ale také k větší jistotě, že testovací prostředí je konzistentní, což v případě předchozího řešení nemusí platit.

4.5 Varianta 2 – Docker

K cíli zmiňovaném ve zhodnocení předchozího řešení vede použití technologií sloužících k vytváření konfiguračních souborů, které jsou vlastně jakýsi předpis pro virtuální stanici a na základě nichž je možné VM rychle vytvořit jako např. Vagrant nebo např. technologie virtualizace na úrovni operačního systému Docker. Tato práce se bude dále věnovat právě kontejnerové virtualizaci Docker, která přináší pro účely testování SW značné výhody.

Kontejnerovou virtualizaci je snadnější si představit více jako virtualizaci procesu než jako ucelenou virtuální stanici, což je zřetelně zobrazeno na Obrázek 21.

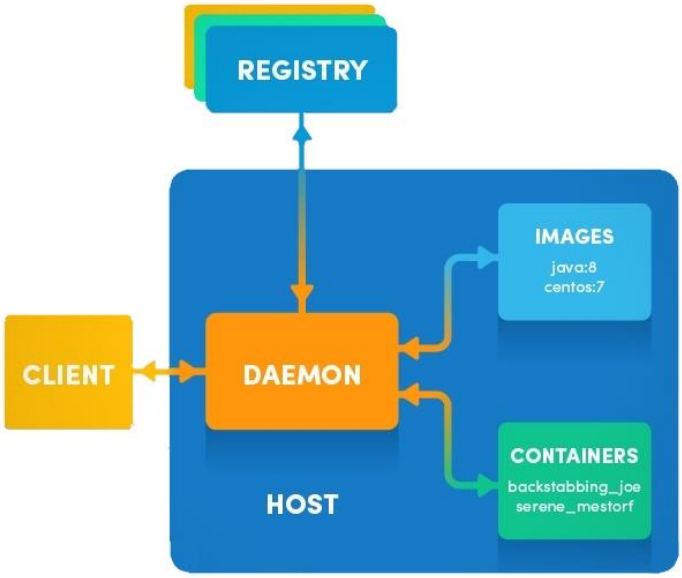
Zatímco u VM s sebou každá virtualizovaná aplikace nese také celý hostovaný operační systém, kontejner běží jako izolovaný proces v uživatelském prostoru (z anglického userspace) hostujícího operačního systému, sdílející funkce jádra s ostatními kontejnery. V důsledku toho snižuje režii klasické virtualizace odstraněním vrstvy zahrnující hostované operační systémy a veškeré úkony s touto vrstvou spojené.



Obrázek 21 – Rozdílné architektury VM a kontejnerů (ZHAW Zurich University of Applied Science, 2014)

4.5.1 Úvod do Dockeru

V rámci práce s Dockerem je potřeba ujasnit některé základní pojmy. Jedná se především o pojmy jako Dockerfile, Docker image a Docker kontejner. Ilustrace práce Dockeru a vzájemné propojení těchto pojmů je dobře vidět na Obrázek 22. Navíc je zde pojem „Registry“, kterým je myšlen online repositář Dockerfilů Docker Hub nebo vlastní lokální repositář.



Obrázek 22 – Ilustrace systému práce Dockeru (Toptal)

4.5.1.1 Dockerfile

Veškerá práce s technologií Docker začíná právě u Dockerfilu, který je uložen v repositáři.

Dockerfile je předpis nebo specifikace toho jak bude vypadat image. Takový soubor obsahuje sled instrukcí definujících jednotlivé příkazy a modifikace výsledného Docker image. Tyto instrukce poskytují značné možnosti nastavení výsledného kontejneru, ale s přidáváním dalších instrukcí roste také velikost výsledného image. Mnoho instrukcí je možné ušetřit využitím již existujícího vyladěného Dockerfilu, který udržuje např. skupina vývojářů určité linuxové distribuce nebo konkrétní aplikace.

Dockerfile je později použit v rámci vytváření Docker image pomocí příkazu *docker build*. Příkaz primárně hledá lokální repositář s Dockerfilem a v případě neúspěchu se ho pokusí najít a posléze stáhnout z tzv. Docker hubu což je veřejný nebo privátní repositář Dockerfilů.

Některé ze základních instrukcí jsou popsány v Tabulka 3.

Instrukce	Funkce
FROM <obraz>	Základní a povinná instrukce určující základní image pro další instrukce.
MAINTAINER <jméno>	Nepovinná položka, která umožňuje nastavit hodnotu poli Author u výsledného image.
RUN <příkaz>	V rámci vytvoření obrazu je pomocí instrukce proveden zadaný příkaz. Tento příkaz je proveden ve vlastní nové vrstvě, která je vzápětí uložena do dalšího obrazu, který je poté použit v dalším kroku. Doporučuje se šetření instrukcemi RUN, právě kvůli dodatečným vrstvám a je doporučené používat možnosti spojování příkazu pomocí &&.
CMD <příkaz>	Instrukce CMD neprovádí žádný příkaz v době vytváření image, ale definuje příkaz, který bude proveden po spuštění kontejneru. Nedochází tedy k vytváření a ukládání nových vrstev.
ENV <klíč>=<hodnota>	Pomocí této instrukce je možné nastavení proměnných prostředí
ADD <zdroj>…<cíl>	Instrukce umožňující zkopírovat adresář či soubory a to jak vzdálené, tak lokální z definovaného zdroje do cíle uvnitř kontejneru. ADD podporuje také komprimovací formáty jako identity, gzip, bzip2 nebo xz a takové soubory jsou případně v rámci instrukce dekomprimovány
COPY <zdroj> <cíl>	V zásadě stejná instrukce jako předchozí ADD, ale neumožňující použití vzdáleného zdroje.
EXPOSE <port>	Informuje Docker, že kontejner má naslouchat na daných síťových portech.
WORKDIR <cesta>	Instrukce nastavující pracovní adresář pro instrukce RUN, CMD, ENTRYPOINT, COPY a ADD
ENTRYPOINT <spustitelný soubor>	Tato instrukce umožnuje nakonfigurovat kontejner jako spustitelný soubor.

Tabulka 3 - Popis základních instrukcí pro vytvoření Dockerfilu (Docker)

4.5.1.2 Docker image

Image, v češtině obraz, je ve své podstatě šablona pro budoucí kontejner. Image je tvořen souborem vrstev dle jednotlivých instrukcí v Dockefilu.

Jakmile je vytvořen Dockerfile, pomocí příkazu *docker build* je z něj vytvořen image a pomocí příkazu *docker run* je pak z image vytvořen běžící kontejner – tedy instance image. Z jednoho Docker image může současně běžet vice jeho instancí.

Image je možné kromě vlastního sestavení i importovat ze souboru pro usnadnění přenositelnosti.

V rámci práce s imagi je nutné představení některé ze základních příkazů viz Tabulka 4. Příkazů pro práci s imagi je mnohem více, ale v rámci práce není možné jejich plné představení.

Příkaz	Funkce
docker images	seznam lokálních imagů
docker rmi	smazání image, nelze provést, běží-li kontejner používající mazaný image
docker search	vyhledání image v repositáři
docker save	uložení image do .tar souboru
docker load	import image z .tar souboru
docker build	sestavení image z Dockerfilu
docker commit	vytvoří image z kontejneru
docker history	zobrazení historie image

Tabulka 4 - Základní příkazy pro práci s Docker imagi (Docker)

4.5.1.3 Docker kontejner

Jakmile je hotový a sestavený Docker image, je již možné vytvořit vlastní běžící kontejner. V mnoha ohledech je provoz kontejneru stejný jako virtuální stanice a pro běžného uživatele zde není velký rozdíl. Pro práci s nimi je ovšem určena sada příkazů, které je dobré znát. Základní příkazy s popisem jejich funkce jsou uvedeny v Tabulka 5.

Příkaz	Funkce
docker ps	seznam běžících kontejnerů, při použití parametru *–a* jsou zobrazeny všechny
docker start	spuštění neběžícího kontejneru
docker stop	vypnutí běžícího kontejneru, odložená vypnutí v sekundách umožňuje parametr -t
docker restart	restart běžícího kontejneru s možností odložení pomocí parametru *–t* v sekundách
docker attach	připojení se k terminálu běžícího kontejneru
docker exec	spuštění příkazu uvnitř běžícího kontejneru
docker inspect	příkaz zobrazující základní informace o kontejneru ve formátu JSON
docker rm	smazání neběžícího kontejneru, pomocí parametru *–f* je možné vynutit smazání i běžícího kontejneru

Tabulka 5 - Základní příkazy pro práci s Docker kontejnery (Docker)

4.5.1.4 Využití Dockeru při řešení daného problému

V rámci řešeného problému je možné Docker použít místo generování nových virtuálních stanic z šablony. Díky Dockeru by tak částečně odpadla také nutnost údržby šablony a nově by bylo potřeba udržovat pouze Dockerfily – tedy pouze kód, který lze bez problémů verzovat.

Výhodou je, že mnoho z potřebných Dockerfilů již existuje v repositářích Docker hubu a je nutné je např. pouze upravit a odladit pro vlastní použití.

Pro praktickou ukázku bude v následující části takto vytvořen, popř. přizpůsoben Dockerfile pro Apache Tomcat 8.

Výraznějším rozdílem při použití Dockeru je, že všechny potřebné artefakty vč. konfiguračních souborů, musí být připraveny před samotnou tvorbou Docker image. U generování virtuálního stroje je ten nejprve spuštěn a až poté je na něj nasazena cílová aplikace a nahrány konfigurační soubory.

Takto upravené pomocné skripty již nebudou součástí práce. Ukázka demonstruje jednoduché nasazení aplikace bez dalších nastavení. V rámci ukázky je využit již existující

plugin do Jenkins CI Docker build step plugin, který přidává do Jenkins jobu možnost provést na nakonfigurovaném Docker hostiteli některý z Docker příkazů.

Dockerfile

Dockerfile pro Apache Tomcat 8 existuje přímo ve veřejném repositáři Docker hubu, ale pro přizpůsobení konkrétnímu účelu je lepší vlastní úprava.

První instrukcí v Dockerfilu je instrukce FROM, která v tomto případě určuje, že základním imagem bude image *java:7-jre*. Vrstvu Javy v imagi tedy není nutné dál řešit.

```
FROM java:7-jre
```

Další instrukce nastavují některé důležité proměnné pro běh Apache Tomcatu jako je vlastní domovská složka CATALINA_HOME a tuto složku přidávají na systémovou PATH. Další potřebnou proměnnou je také URL, ze kterého bude v jednom z dalších kroků stažen instalační balíček Tomcatu.

```
ENV CATALINA_HOME /usr/local/tomcat
ENV PATH $CATALINA_HOME/bin:$PATH
ENV TOMCAT_TGZ_URL http://archive.apache.org/dist/tomcat/tomcat-
8/v8.0.30/bin/apache-tomcat-8.0.30.tar.gz
```

Následuje nastavení domovské složky Tomcatu jako aktuální pracovní adresář.

```
WORKDIR $CATALINA_HOME
```

Další krok se liší od oficiálního Dockerfilu poskytovaném Tomcatem.

Jedním z požadavků je možnost pozdějšího přístupu na běžící kontejner pomocí SSH, ale protože kontejnery jsou primárně koncipované jako jednoprocesové, je potřeba nainstalovat také aplikaci supervisor, který umožnuje spuštění více souběžně běžících procesů v jednom kontejneru.

Dalším příkazem je také nastavení hesla k připojení pomocí ssh na ‘test’ a povolení vzdáleného root loginu v konfiguraci SSH. Tyto všechny příkazy jsou spojeny operátorem && tak, aby nebyly zbytečně přidány další vrstvy do Docker image.

```
RUN apt-get update && apt-get install -y supervisor openssh-server mc &&
mkdir /var/run/sshd && echo 'root:test | chpasswd && sed -i
's/PermitRootLogin without-password/PermitRootLogin yes/'
/etc/ssh/sshd_config
```

Správná funkce programu supervisor vyžaduje existenci konfiguračního souboru supervisord.conf. V rámci sestavení image je tedy potřeba předpřipravený soubor zkopírovat do správné lokace.

```
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
```

Následující instrukce RUN je opět spojení několika příkazů, které mají postupně stáhnout instalační balíček Tomcatu z dříve nadefinovaného URL, tento balíček rozbalit a poté ho smazat.

```
RUN curl -fSL "$TOMCAT_TGZ_URL" -o tomcat.tar.gz \
    && tar -xvf tomcat.tar.gz --strip-components=1 \
    && rm bin/*.bat && rm tomcat.tar.gz*
```

Další instrukce COPY už kopíruje .war soubor s aplikací do složky webapps určené k nasazování aplikací.

```
#Copy test.war
COPY test.war webapps/
```

Instrukce EXPOSE dává Dockeru instrukci, že je v budoucnu vytvořený kontejner bude poslouchat na portech 22 pro SSH spojení a 8080 webový server Apache Tomcat.

```
#Expose port and run it
EXPOSE 22 8080
```

Poslední instrukce CMD je instrukce zabezpečující, že ve výsledném kontejneru se po spuštění spustí právě program supervisord, který pomocí konfigurace spustí další procesy.

```
CMD ["/usr/bin/supervisord"]
```

Konfigurační soubor programu supervisor, který je spuštěn po startu kontejneru, vypadá následovně:

```
[supervisord]
nodaemon=true

[program:test-app]
directory=/usr/local/tomcat/bin/
command=/usr/local/tomcat/bin/startup.sh

[program:sshd]
command=/usr/sbin/sshd -D
```

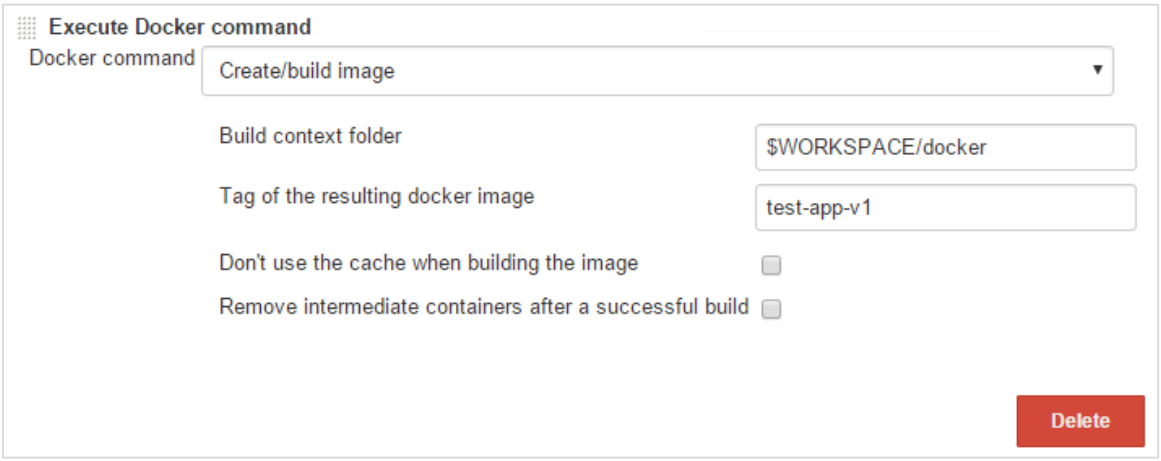
Jak je vidět výše, tento konfigurační soubor obsahuje pro každý program, který má na cílovém kontejneru běžet, nadefinované složky, ve kterých se nachází jednotlivé programy ke spuštění a také příkazy, kterými je tyto programy možné nastartovat.

Jenkins job

V okamžiku, kdy je hotový samotný Dockerfile, je již možné pokusit se sestavit image a následně z něj nastartovat kontejner. To je možné buď přímo z příkazové řádky na hostovské stanici s Dockerem nebo z Jenkins CI pomocí pluginu. Plugin Docker build step plugin je možné přidat jako dodatečný Build step.

Pro každou operaci je potřeba přidat další Build step a nastavit potřebné parametry.

Pro vytvoření image je nutné nastavit složku, ve které se má hledat Dockerfile a všechny potřebné soubory a jméno výsledného image jak ukazuje Obrázek 23.



Obrázek 23 - Docker build step plugin - vytvoření image z Dockefulu

Odpovídající docker příkaz by vypadal následovně.

```
docker build $WORKSPACE/docker/ -t test-app-v1
```

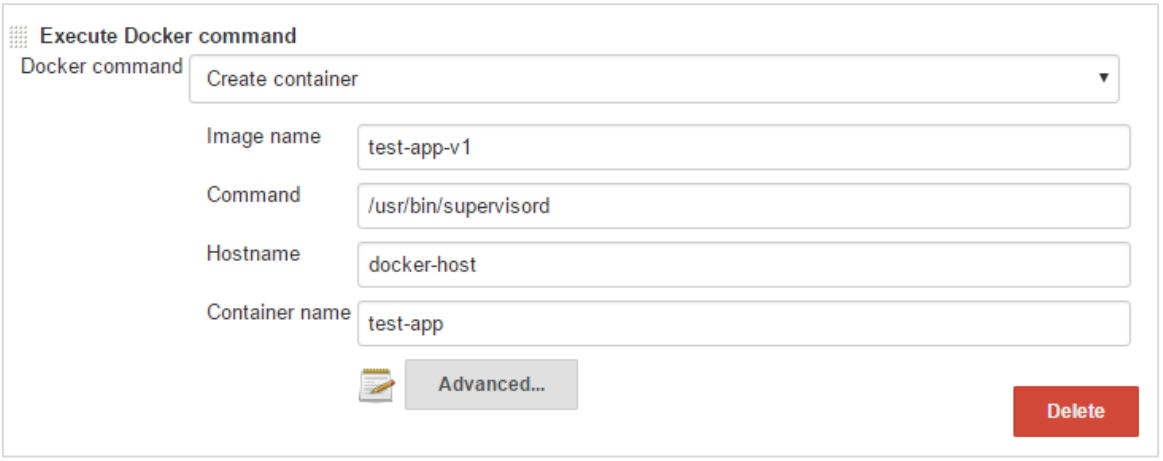
Po provedení příkazu je možné výsledný image zkontrolovat příkazem docker images, který vypíše všechny v systému přítomné sestavené image.

```
root@docker-host:~# docker images
REPOSITORY        TAG         IMAGE ID          CREATED    VIRTUAL SIZE
test-app-v1      latest      b440ca939b37     1 hour ago    560.5 MB
java              7-jre      16ad6c7dbc4c     1 hour ago    343.7 MB
```

V tomto případě by výpis ukázal dva image. První by byl image z instrukce FROM a druhý by byl sestavený požadovaný image.

V okamžiku, kdy již existuje image, je z něj možné v dalším kroku vytvořit kontejner. Ukázka nastavení pluginu pro vytvoření kontejneru z image z minulého kroku je vidět na

Obrázek 24, kde je již nadefinováno také jméno výsledného kontejneru a příkaz, který se na něm spustí.

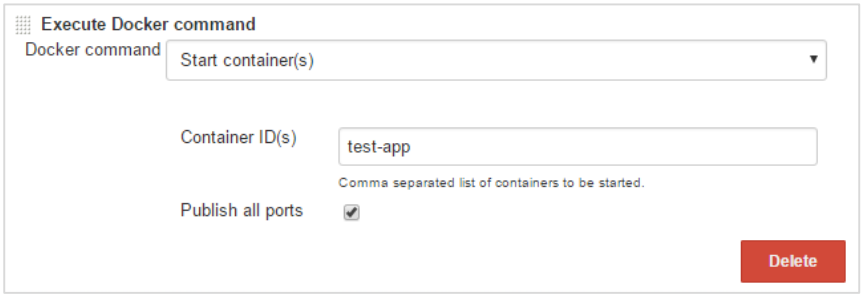


Obrázek 24 - Docker build step plugin - vytvoření kontejneru z existujícího image

Po tomto kroku je již připravený kontejner, ale neběží. Tento krok se obvykle nepoužívá a kontejner se vytváří a pouští v jednom kroku.

Příkaz pro pouhé vytvoření by vypadal takto:

```
docker create -P --name="test-app" test-app-v1 /usr/bin/supervisord
```



Obrázek 25 - Docker build step plugin - spuštění již vytvořeného kontejneru

Při použití Docker build step pluginu je nutné pro samotné spuštění kontejneru ještě další build step viz. Obrázek 25. Zde je potřeba nastavit pouze jméno kontejneru vytvořeného v předchozím kroku. Důležité je také nastavení „Publish all ports“, které namapuje všechny porty z instrukce EXPOSE v Dockerfilu na porty Docker hostitele, přes které jsou pak tyto porty přístupné. Příslušný příkaz pro spuštění kontejneru by mohl vypada např. takto:

```
docker start -i test-app
```

Ve výpise běžících kontejnerů, který je možné získat pomocí příkazu *docker ps*, je pak vidět jméno kontejneru, jeho ID, jméno image ze kterého pochází, spuštěný příkaz a také mapování portů. V tomto konkrétním výpisu je port 8080 v kontejneru namapován na port 32894 na hostovském Docker serveru (v rámci příkladu server s názvem docker-host) a port 22 pro připojení přes SSH je namapován na port 32895. Testovaná aplikace by tedy měla být přístupná např. přes adresu *http://docker-host:32894/test*.

```
root@docker-host:~# docker ps
ID     IMAGE         COMMAND                              PORTS                  NAMES
36de test-app-v1 "/usr/bin/supervisord" 32895->22/tcp, 32894->8080/tcp test-app
```

Kompletní průběh vytvoření image a vytvoření a následné spuštění kontejneru je také dobře vidět v logu Jenkins jobu viz Příloha 1. Je zde patrné rozdělení vytváření image do jednotlivých kroků, které pak také tvoří jeho jednotlivé vrstvy.

5 Zhodnocení výsledků a doporučení

V úvodní teoretické části byly představeny principy kontinuální integrace, které již dnes vnímá jako nedílnou součást vývoje SW, většina vývojářů. Jedním z dále představených základních principů kontinuální integrace je automatické testování, které se stále podceňuje. Jedním z důvodů může být také náročnost na zajištění potřebné testovací infrastruktury. Právě díky dále představené virtualizaci je zajištění infrastruktury mnohem snazším úkolem než dříve.

V rámci řešení praktické úlohy byly představeny dvě varianty řešení.

První varianta představila sadu skriptů a počítala s přednastavenou šablonou virtuální stanice.

Druhá varianta využila kontejnerovou virtualizační technologii Docker, která mohla v představené sadě skriptů nahradit skupinu skriptů generující novou virtuální stanici.

Varianta s využitím Dockeru se jeví jako vhodnější i rychlejší. Docker je ovšem nová technologie a ještě do nedávna nebyla doporučena pro použití v produkčním prostředí, což je jeden z důvodů proč s ní nebylo počítáno hned v první variantě. Dalším důvodem byla také chybějící možnost integrace Dockeru přímo do XenServeru. Tato podpora je od verze XenServer 6.5 již dostupná.

Celý průběh od vytvoření image až po spuštění Apache Tomcat vč. nasazení testované aplikace trval kolem jedné a půl minuty, což je minimálně dvakrát rychlejší průběh než u předchozí varianty. Tato varianta také umožňuje naprostou izolovanost jednotlivých konfigurací, konzistentní prostředí a v neposlední řadě možnost přenositelnosti image nebo kontejneru. Kdokoli by měl k dispozici Dockerfile a všechny soubory v něm potřebné, může si Docker kontejner spustit ve vlastním prostředí.

V neposlední řadě je velice výhodné, že v případě varianty s Dockerem je prakticky infrastruktura v kódu (Dockerfile), který je snadné verzovat.

6 Závěr

Neustálý růst a vznik nové konkurence vyvíjí na firmy v celém IT odvětví tlak na inovativnost a efektivitu při stávající či dokonce vyšší kvalitě. Tento tlak není celkově špatný, protože má za následek velké množství ušetřených peněz, redukci chyb a incidentů a rostoucí kvalitu produktů.

Vývoj v týmu, který se plně drží zásad kontinuální integrace je rychlejší a tím i levnější. Díky pravidelnému automatickému testování v jsou navíc defekty odhaledny v dřívějších fázích vývoje. Náklady na opravu takových chyb jsou v počátečních fázích vývoje nesrovnatelně nižší než by byly v případě jejich nalezení až v době nasazení produktu u koncového zákazníka.

Předpokladem pro kvalitní testování je ale existence kvalitní infrastruktury. V této oblasti je trendem posledních let virtualizace a automatizace. Zatímco výpočetní výkon dedikovaných serverů je zřídkakdy využíván z více než 10 %, díky virtualizaci je možné využít plný výkon serveru. Úsporu tedy virtualizace nepřináší pouze při vlastním nákupu potřebného hardwaru, ale také ve spotřebě energie nebo množství personálu pořebného k jeho údržbě.

Časem se nicméně ukázalo, že i virtualizovaná infrastruktura nebezpečně bobtná a vyžaduje další nemalé investice. Pro každou firmu je nezbytné vyhýbat se možným prodlevám v produkci a za preventivní opatření v podobě zálohovaích řešení či řešení tzv. disaster recovery, jsou firmy ochotny utratit nemalé částky. Směrem, který se snaží minimalizovat tyto náklady se vyvíjí trendy v automatizaci.

V případě řešeného problému, tedy konkrétně v případě automatizace vytváření testovací infrastuktury je jasným benefitem časová úspora z původní hodiny na necelou minutu. Kromě časové úspory je však největším benefitem právě nenáročnost automatizované infrastruktury na údržbu. V případě druhé varianty je infrastruktura kompletně přepsána do kódu a je možné ji kdykoli znovu uvést do provozu bez zmíněných dodatečných nákladů. Investice do zálohovacích řešení tak najednou ztrácí smysl.

Je vidět, že je stále možné najít mnoho možností jak v celém IT odvětví šetřit zdroje a zefektivňovat práci. V příštích letech bude jistě zajímavé sledovat další vývoj v představené oblasti.

7 Seznam použité literatury

AbcLinuxu. 2011. Kontinuální integrace s Jenkins CI. *AbcLinuxu.* [Online] AbcLinuxu, 31. 07 2011. [Citace: 2016. 2 10.] http://www.abclinuxu.cz/blog/lojzoviny/2011/7/kontinualni-integrace-s-jenkins-ci.

Borovcová, Anna. 2008. *Část II: Základy testování.* 2008.

Bourque, Alain Abran a Pierre. 2004. *Guide to the software engineering body of knowledge.* Los Alamitos : IEEE Computer Society, 2004. ISBN 0769523307.

Bureš, Michal. 2009. *Virtualizace - porovnání dvou daných platforem.* Praha : autor neznámý, 2009.

Citrix. Citrix Unveils End-to-End Virtualization Strategy. *Citrix.* [Online] Citrix. [Citace: 10. 2 2016.] https://www.citrix.com/go/news/xensource.html.

—. 2015. XenServer 6.5: Technical FAQ. *Support Citrix.* [Online] 2015. [Citace: 2. 10 2016.] http://support.citrix.com/content/dam/supportWS/kA560000000Ts7qCAC/XenServer_6.5.0_Technical_FAQ.pdf.

Docker. Dockerfile reference. *Docker.* [Online] [Citace: 20. 2 2016.] https://docs.docker.com/engine/reference/builder/.

Everything VM. 2011. History of Virtualization. *Everything VM.* [Online] 8. 1 2011. [Citace: 23. 1 2016.] http://www.everythingvm.com/content/history-virtualization.

Fowler, Martin. 2006. martinfowler.com. *Continuous Integration.* [Online] 2006. [Citace: 22. 12 2015.] http://www.martinfowler.com/articles/continuousIntegration.html.

Free Software Foundation. GNU Operating System. *GNU Make.* [Online] Free Software Foundation. [Citace: 22. 2 2016.] https://www.gnu.org/software/make/.

Gigaom. The sorry state of server utilization and the impending post-hypervisor era. *Gigaom.* [Online] [Citace: 21. 2 2016.] https://gigaom.com/2013/11/30/the-sorry-state-of-server-utilization-and-the-impending-post-hypervisor-era/.

Hailpern, Brent a Santhanam, Padmanabhan. 2001. *Software Debugging, Testing, and Verification.* New York : IBM Research Division, 2001.

Hujer, Martin. 2012. *Bakalářská práce: Kontinuální integrace při vývoji webových aplikací v PHP*. Praha : Vysoká škola ekonomická v Praze, 2012.

IBM. DB2® object naming rules. *IBM Knowledge Center.* [Online] IBM. [Citace: 12. 2 2016.] http://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.7.0/com.ibm.db2.luw.admin.dbobj.doc/doc/c0007246.html.

Jenkins. 2016. Plugins. *Jenkins CI.* [Online] 2016. [Citace: 2016. 2 10.] https://wiki.jenkins-ci.org/display/JENKINS/Plugins.

Jenkins, Nick. 2008. *A Software Testing Primer.* San Francisco : Creative Commons, 2008. An Introduction to Software Testing.

Kábrt, Tomáš. 2012. *Virtualizace operačních systémů v serverech.* Praha : České vysoké učení technické v Praze, 2012.

Matyska, Luděk. *Techniky virtualizace počítačů.* místo neznámé : ÚVT MU. Virtualizace výpočetního prostředí. ISSN 1212-0901.

National Instruments. 2014. NI Real-Time Hypervisor Architecture and Performance Details. *National Instruments.* [Online] 10. 12 2014. [Citace: 8. 2 2016.] http://www.ni.com/white-paper/9629/en/.

Nový, Lukáš. 2007. *Porovnání virtualizačních technik.* Brno : Msarykova univerzita, 2007.

Oracle. Brief History of Virtualization. *Oracle.* [Online] Oracle. [Citace: 24. 1 2016.] https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html.

—. 2012. Hypervisor. *Introduction to Virtualization.* [Online] 2012. [Citace: 8. 2 2016.] https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1011.html.

Pavlis, Martin a Vávra, Jan. 2009. Možnosti virtualizace, přínosy virtualizace serverů. *System online.* [Online] IT SYSTEMS, 6 2009. [Citace: 23. 1 2016.]

http://www.systemonline.cz/virtualizace/moznosti-virtualizace-prinosy-virtualizace-serveru.htm.

pepgotesting.com. http://www.pepgotesting.com/. *Automated software testing in Continuous Integration (CI) and Continuous Delivery (CD).* [Online] [Citace: 22. 12 2015.] http://www.pepgotesting.com/continuous-integration/.

Pietrik, Michal. 2012. *Diplomová práce: Automatizované testování webových aplikací.* Brno : Masarykova univerzita, 2012.

Šolc, Robin. 2007. *Bakalářská práce: Nástroje pro automatické testovíní softwaru.* Praha : Vysoká škola ekonomická v Praze, 2007.

tldp.org. $RANDOM: generate random integer. *Advanced Bash-Scripting Guide:*. [Online] [Citace: 12. 2 2016.] http://tldp.org/LDP/abs/html/randomvar.html.

Tomášek, Patrik. 2015. *Virtualizace serverů a aplikací pomocí Dokcer.* Praha : autor neznámý, 2015.

Toptal. Getting Started with Docker: Simplifying Devops. *Toptal.* [Online] [Citace: 21. 2 2016.] http://www.toptal.com/devops/getting-started-with-docker-simplifying-devops.

VMware, Inc. 2007. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist.* [PDF Document] Palo Alto : VMware, Inc., 2007.

—. Virtualization. *VMware.* [Online] [Citace: 24. 1 2016.] https://www.vmware.com/files/pdf/GATED-VMW-EBOOK_VIRTUALIZATION-ESSENTIALS.pdf.

Wikimedia Commons. 2011. Wikimedia Commons. *File:SVN Server Client Structure.png.* [Online] 2011. [Citace: 23. 12 2015.] https://commons.wikimedia.org/wiki/File:SVN_Server_Client_Structure.png#globalusage.

Wikipedia. Ring (CPU). *Wikipedia.* [Online] Wikipedia. [Citace: 31. 1 2016.] https://de.wikipedia.org/wiki/Ring_(CPU).

Zamborský, Bc. Matúš. 2012. *Diplomová práca: Kontinuálna integrácia.* Brno : Masarykova univerzita, 2012.

ZHAW Zurich University of Applied Science. 2014. ICCLAB SPLAB. *ICCLab News.* [Online] 6 2014. [Citace: 16. 2 2016.] https://blog.zhaw.ch/icclab/icclab-news-july-2014-issue-n-2/.

8 Seznam použitých obrázků

9 Seznam tabulek

10 Přílohy

Příloha 1 - Jenkins job log

```
[Docker] INFO: Creating docker image from /jenkins/jobs/test-app-
v1/workspace/docker
Step 1 : FROM java:7-jre
Pulling from library/java
Status: Downloaded newer image for java:7-jre
 ---> 16ad6c7dbc4c
Step 2 : ENV CATALINA_HOME /usr/local/tomcat
 ---> Using cache
 ---> 1ceb956a1b1e
Step 3 : ENV PATH $CATALINA_HOME/bin:$PATH
 ---> Using cache
 ---> 683260c8d073
Step 4 : ENV TOMCAT_TGZ_URL http://archive.apache.org/dist/tomcat/tomcat-
8/v8.0.30/bin/apache-tomcat-8.0.30.tar.gz
 ---> Using cache
 ---> 71ae09065c93
Step 5 : WORKDIR $CATALINA_HOME
 ---> Using cache
 ---> 371f978bb845
Step 6 : RUN apt-get update && apt-get install -y supervisor openssh-server mc
&& mkdir /var/run/sshd && echo 'root:test' | chpasswd && sed -i
's/PermitRootLogin without-password/PermitRootLogin yes/' /etc/ssh/sshd_config
 ---> Using cache
 ---> 48ca8d8050b4
Step 7 : COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
 ---> Using cache
 ---> 0eb94c3239f2
Step 8 : RUN curl -fSL "$TOMCAT_TGZ_URL" -o tomcat.tar.gz && tar -xvf
tomcat.tar.gz --strip-components=1  && rm bin/*.bat && rm tomcat.tar.gz*
 ---> Using cache
 ---> 24bb1dadc9a6
Step 9 : COPY test.war webapps/
 ---> b6384df6357e
Removing intermediate container e08791fc0331
Step 10 : EXPOSE 22 8080
 ---> Running in 1957bc185f5a
 ---> be10f8be7b51
Removing intermediate container 1957bc185f5a
Step 11 : CMD /usr/bin/supervisord
 ---> Running in 1b88851f0605
 ---> b440ca939b37
Removing intermediate container 1b88851f0605
Successfully built b440ca939b37
[Docker] INFO: Sucessfully created image test-app-v1
[Docker] INFO: created container id test-app (from image test-app-v1)
[Docker] INFO: started container id test-app
```