VYSOKÁ ŠKOLA EKONOMICKÁ V PRAZE

Fakulta informatiky a statistiky

Katedra informačních technologií

**BAKALÁŘSKÁ PRÁCE**

2012 Martin Hujer

**Vysoká škola ekonomická v Praze**

**Fakulta informatiky a statistiky**

**Katedra informačních technologií**

Studijní program: Aplikovaná informatika

Obor: Informatika

**Kontinuální integrace při vývoji webových aplikací v PHP**

**BAKALÁŘSKÁ práce**

Student : Martin Hujer

Vedoucí : Ing. Jan Mittner

Oponent : @tdo Jméno a příjmení s tituly

**2012**

**Prohlášení**

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne @tdo den. měsíc 2012 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Martin Hujer

**Poděkování**

Zde je prostor pro vyjádření poděkování… @tdo ………………………….

**Abstrakt**

Cílem BP je připravit text, který umožní např. malému webovému studiu získat nezbytné teoretické znalosti pro implementaci CI platformy pro PHP a zároveň ho provede její praktickou implementací.

//@tdo Možná zkusit podle draftu nasadit ve Vergiliu, napsat o tom do BP

Obsahuje zaměření a hlavní cíl práce, způsob dosažení cíle, přínos práce (vlastní příspěvek k řešenému tématu) a stručně popsanou strukturu práce. @tdo

**Klíčová slova**

Seznam nejvýznamnějších odborných výrazů charakterizujících téma závěrečné práce. @tdo

**Abstract**

Anglická verze abstraktu. @tdo

**Keywords**

seznam klíčových slov v anglickém jazyce. @tdo

@tdo vygenerovat obsah //až ke konci, jinak má Word problémy

# Úvod

@todo Webové aplikace se dnes rozvíjejí a mění rychle, takže je potřeba sledovat a kontrolovat kvalitu kódu.

Při vývoji softwarového projektu většinou jednotliví vývojáři nebo jednotlivé vývojářské týmy pracují na samostatných subsystémech, které je poté nutné propojit dohromady. Tento proces se označuje jako integrace. Tradiční metodiky vývoje software jako je například Vodopádový model zařazují fáze integrace, testování a kontroly kvality až po dokončení fáze vývoje (viz Obrázek 1.1). Doba samotného vývoje software je často odhadnuta nesprávně, i když tomu lze částečně předejít. [McConnell, 2006]

//@todo najít knížky od Voříška/Buchalcevové a z nich vzít obrázek a citaci waterfallu a Scrumu

Obrázek 1.1 Schéma vodopádového modelu vývoje software

Odhad doby trvání integrační a QA[[1]](#footnote-1) fáze je vzhledem k jejich neurčitosti prakticky nemožný. Fáze s neznámou dobou trvání v závěru projektu, kdy už ostatní fáze mohly překročit odhadované doby trvání, může způsobit překročení plánované doby trvání celého projektu a tedy i rozpočtu.

S řešením problému neurčitosti integrační fáze přichází technika tzv. kontinuální (neboli průběžné) integrace. Fáze integrace, testování a kontroly kvality se rozloží do celého průběhu fáze implementace a jsou prováděny průběžně, vždy po každé změně programového kódu. Kontinuální integrace je často využívána současně s tzv. agilními metodikami vývoje, kdy je software vyvíjen v krátkých iteracích.

//@todo tady bude schéma scrumu

# Rešerše zdrojů pojednávajících o kontinuální integraci

@todo

# Charakteristika konceptu kontinuální integrace

Kontinuální integrace zavádí proces průběžné kontroly kvality kódu – často prováděné nenáročné úkony, místo kontroly kvality až po dokončení vývoje.[[2]](#footnote-2)

[Fowler, 2006] definici dále upřesňuje:

Kontinuální integrace je technika používaná při vývoji software, kdy členové týmu integrují svůj kód často, obvykle aspoň jednou za den, což vede k mnoha integracím každý den. Každá integrace je ověřená automatickým sestavením (včetně testů), aby byly případné problémy objeveny co nejdříve. Mnoho týmů zjistilo, že tento přístup vede k výraznému snížení problémů při integraci a umožňuje jim rychleji dodávat kvalitnější software.[[3]](#footnote-3)

Kontinuální integrace byla poprvé popsána[[4]](#footnote-4) jako jedna z praktik tzv. Extrémního programování[[5]](#footnote-5), nicméně je ji samozřejmě možné využít i bez implementace ostatních praktik EP.

## Přehled technik a zásad CI

Kontinuální integrace je souhrnem několika technik, které lze do týmového workflow zahrnout samostatně (a tedy postupně). V této kapitole a jejích podkapitolách shrnu jednotlivé techniky podle [Fowler, 2006] a [Duvall, et al., 2007].

### Jednotné úložiště zdrojových kódů

Každý softwarový projekt se skládá z mnoha souborů různých typů – soubory se zdrojovým kódem, konfigurační soubory, multimediální data (například obrázky, které jsou součástí webu). Při spolupráci více lidí na projektu může být problém zajistit, aby všichni měli všechny soubory v aktuální verzi. Dříve bylo běžné na sdílení projektových souborů používat sdílené síťové disky, ale toto řešení neumožňuje snadno získat starší verzi souboru se zdrojovým kódem.

Tento problém řeší nástroje pro správu zdrojového kódu[[6]](#footnote-6), které kromě úložiště (Repository) nabízejí i přístup ke starším verzím souborů. Úložiště může být buď centrální (VCS) anebo distribuované (DVCS). Mezi nejznámější open-source nástroje pracující s centrálním úložištěm patří Subversion[[7]](#footnote-7) nebo dnes již zastaralé CVS[[8]](#footnote-8). Z komerčních lze jmenovat Rational Team Concert[[9]](#footnote-9) od IBM nebo Team Foundation Server[[10]](#footnote-10) od společnosti Microsoft. Mezi nejznámější open-source nástroje pracující s distribuovaným úložištěm patří Git[[11]](#footnote-11) (je používaný pro verzování jádra operačního systému Linux), Mercurial[[12]](#footnote-12) nebo Bazaar[[13]](#footnote-13). Z komerčních lze uvést BitKeeper[[14]](#footnote-14) od společnosti BitMover, Inc. (používaný pro verzování jádra operačního systému Linux v letech 2002-2005[[15]](#footnote-15)).

Jakmile tým používá jednotné úložiště zdrojového kódu, odpadají problémy s dohledáváním chybějících souborů. To předpokládá, že do repositáře se ukládá opravdu vše – i konfigurační soubory, databázová schémata, instalační skripty a knihovny třetích stran. [Fowler, 2006] doporučuje jednoduché pravidlo: Pokud na čistě nainstalovaném počítači provedete checkout[[16]](#footnote-16) repositáře, tak by mělo být možné provést sestavení aplikace[[17]](#footnote-17). Výjimku tvoří například operační systém, JDK[[18]](#footnote-18) nebo databázový systém. Fowler také doporučuje do úložiště ukládat i další soubory, které nejsou nezbytně nutné k sestavení aplikace, ale mohou ušetřit práci ostatním členům týmu – například konfigurační soubory pro IDE[[19]](#footnote-19).

Nicméně například [Menšík, 2009] s Fowlerem nesouhlasí a považuje ukládání knihoven do verzovacího systému za zbytečné. Zároveň ale zmiňuje nutnost umožnit skriptu pro sestavení přístup ke sdílenému síťovému disku s knihovnami.

V úložišti by mělo být uloženo vše, co je potřeba pro sestavení aplikace, ale nic z toho, co vznikne při sestavení – podle Fowlera to většinou znamená, že není snadné provést spolehlivé sestavení aplikace.

### Automatizované sestavení aplikace

Převod zdrojových kódů do funkční aplikace je často složitý proces, který zahrnuje kompilaci zdrojových kódů, přesouvání souborů, nahrávání schémat do databáze a další. Vzhledem k tomu, že tento proces lze automatizovat, tak by také automatizovaný měl být. Pokud ho provádí člověk ručně, tak zbytečně ztrácí čas a může se stát, že udělá chybu.

Mezi systémy pro automatizované sestavení aplikace lze zařadit make[[20]](#footnote-20) používaný v operačních systémech Linux a Unix, Ant[[21]](#footnote-21) používaný pro projekty psané v jazyce Java nebo Phing[[22]](#footnote-22) používaný pro projekty psané v jazyce PHP.

Častou chybou je, že automatizovaný build nezahrnuje všechny nutné kroky a je nutné manuálně provést některé další úkony po skončení automatizovaného sestavení. Fowler zde doplňuje svoje pravidlo: Pokud na čistě nainstalovaném počítači provedete checkout repositáře, tak by mělo být možné pomocí jednoho příkazu získat běžící aplikaci.

IDE často obsahují nástroje na provádění sestavení, ale jejich použití je problematické, protože soubory s popisem procesu sestavení jsou většinou proprietární pro dané IDE a není je možné bez něj spustit. Samozřejmě je možné, aby jednotliví vývojáři spouštěli lokální buildy přímo z IDE, ale vždy potřeba mít skript pro sestavení, který lze spustit na serveru a je nezávislý na IDE.

#### Upřesnění pro projekty vyvíjené v jazyce PHP

Skripty psané v jazyce PHP se většinou nekompilují předem, ale až při zpracování v interpreteru. Proto skript pro sestavení aplikace nebude, na rozdíl například od projektů psaných v jazyce Java, obsahovat kompilaci, ale jen kontrolu syntaktické správnosti zdrojových kódů.

### Testovatelný build

Úspěšné sestavení neznamená, že aplikace bude fungovat správně. Pro odchycení chyb je vhodné do procesu sestavení zařadit i automatizované testování. Vhodným, nicméně ne nezbytným, přístupem je proto Test Driven Development[[23]](#footnote-23), kdy se nejdříve píše test a teprve poté kód aplikace. TDD je jednou z klíčových praktik EP [Beck, 2005].

Nejčastěji používanými testovacími nástroji jsou tzv. xUnit[[24]](#footnote-24) frameworky, kde x značí programovací jazyk. Existuje tedy JUnit[[25]](#footnote-25) pro Javu, NUnit[[26]](#footnote-26) pro .NET, PHPUnit[[27]](#footnote-27) pro PHP a další.

Testy jsou většinou sdružovány do sad testů a po jejich spuštění je vygenerován report, který informuje o tom, zda některé testy selhaly.

[Fowler, 2006] upozorňuje na to, že testy neobjeví všechny chyby. Nicméně nedokonalé testy, které jsou spouštěny často, jsou podle něj mnohem lepší než dokonalé testy, které nejsou nikdy napsány.

### Každý vývojář ukládá kód do úložiště alespoň jednou za den

Nejsnazším způsobem, jak může dát vývojář vědět ostatním kolegům v týmu o změnách v nějaké části aplikace je uložení změn do úložiště. Do úložiště je samozřejmě vhodné ukládat jen funkční kód, což nutí vývojáře rozdělit velký úkol na mnoho dílčích podúkolů, které lze řešit samostatně. To, že je kód funkční, ověří vývojář sestavením aplikace a spuštěním automatizovaných testů. Těsně před uložením změn si musí vývojář stáhnout případné změny, které provedli ostatní členové týmu a testy ověřit, že jeho kód stále funguje i s těmito změnami. Teprve poté může kód uložit.

Pokud by při stažení nejnovějších změn z úložiště narazil na nějaký konflikt, tak většinou nebude problém ho vyřešit, protože během těch několika hodin od minulého stažení nemohlo změn být mnoho.

### Po každém commitu proběhne sestavení aplikace na integračním stroji

Některé společnosti provádějí sestavení aplikace v naplánovaných intervalech (např. každou noc), ale to není úplně ideální, protože cílem kontinuální integrace je odhalit chyby co nejdříve je to možné. Pokud sestavení probíhá jednou za 24 hodin, může se stát, že chyby v systému zůstanou dlouhou dobu (a budou působit problémy ostatním členům týmu). Proto je mnohem lepší provádět sestavení po každé změně v úložišti. To může probíhat buď manuálně, nebo automaticky.

Při manuálním sestavení si vývojář po uložení změn do úložiště sedne k speciálnímu počítači, který je vyhrazen jen na integraci[[28]](#footnote-28). Stáhne si aktuální verzi z úložiště, spustí sestavení, a pokud to proběhne správně, tak je jeho úkol hotový.

Druhou variantou je využití integračního serveru, který sleduje úložiště (v pravidelných intervalech se dotazuje úložiště – tzv. polling), a po každé zjištěné změně spustí build a pošle autorovi e-mail s výsledkem.

Integrační server není nezbytně nutný, ale jeho použití je pohodlnější. Jeho další výhodou je, že pokud součástí buildu je i statická analýza zdrojového kódu[[29]](#footnote-29), můžeme na výstupu CI serveru sledovat dlouhodobé trendy v kvalitě kódu.

### Sestavení musí být rychlé

Cílem kontinuální integrace je rychle poskytnout vývojářům zpětnou vazbu. Proto by sestavení na integračním serveru mělo být co nejrychlejší. Kent Beck [Beck, 2005] ve své metodice extrémního programování doporučuje, aby build netrval déle než 10 minut. Pokud by trval déle, výrazně se zvyšuje šance, že ho vývojáři nebudou spouštět jako lokální build.

Často je sestavení zpomalováno testy, které pracují s externími zdroji, jako je databáze nebo služby. V takovém případě je možné build rozdělit na dvě části – primární (spouští se po commitu 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 build zjistil nějaké problémy, tak to není tak velký problém, protože testy důležité funkčnosti jsou zahrnuté v primárním sestavení.

Pokud by sekundární build trval moc dlouho, tak 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.

### Testování by se mělo provádět v prostředí co nejpodobnějším produkčnímu

Cílem testování je odhalit problémy, které by mohly nastat v produkčním prostředí. Proto je vhodné, aby testovací prostředí bylo co nejvíce podobné produkčnímu. Pokud se budou lišit, tak se může stát, že se některé chyby nepodaří odhalit nebo naopak bude docházet k planým poplachům.

Například pokud by vývojáři pracovali na počítačích s operačním systémem Microsoft Windows, integrační server by také používal Windows, ale na produkčním serveru by aplikace běžela na operačním systému Linux, neprojevily by se problémy s rozdílnými znaky pro konce řádků nebo oddělovače adresářů v cestě.

Proto by integrační server měl běžet na stejném operačním systému, používat stejné verze aplikací, běhových prostředí i knihoven.

### Každý má snadný přístup k informacím

Při aplikaci kontinuální integrace je důležitá komunikace mezi členy týmu. Jednou z nejdůležitějších informací je stav posledního sestavení, protože pokud to bylo neúspěšné (což značí, že kód obsahuje chybu nebo chyby), tak není vhodné, aby si vývojář aktualizoval svoji kopii zdrojových kódů z úložiště na tuto verzi.

Pokud se využívá manuální integrace na integračním stroji, tak by na jeho monitoru mělo vždy být vidět stav posledního sestavení.

Automatické nástroje pro kontinuální integraci umožňují snadnější přístup k této informaci. Členové geograficky distribuovaných vývojových týmů mohou dostávat tuto informaci pomocí e-mailu, SMS, aplikace běžící na počítači vývojáře nebo pomocí pluginu do IDE. Týmy často experimentují s vizuálními pomůckami, které ukazují aktuální stav buildu. Některé používají například červenou a zelenou lávovou lampu[[30]](#footnote-30), LED diody řízení Arduinem[[31]](#footnote-31) nebo detailnější tabuli s přehledem stavu buildů[[32]](#footnote-32).

### Automatizované nasazení

Využívání kontinuální integrace umožňuje častěji získávat stabilní sestavení aplikace, čehož lze využít pro zvýšení frekvence vydávání a nasazování nových verzí. Tato technika se označuje jako Continuous Delivery. Pokud by systém automaticky nasadil do provozu každé úspěšné sestavení, které projde automatizovanými testy, označovali bychom to jako Continuous Deployment.

Podrobný rozbor tématu však přesahuje rozsah této práce. Více informací lze nalézt například v [Humble, a další, 2010], případně v [Zikmund, 2011] se zaměřením jen na PHP.

## Typický průběh práce vývojáře v prostředí CI

1. Vývojář si z úložiště stáhne nejnovější verzi zdrojových kódů
2. Provede potřebné úpravy - většinou trvají jen několik hodin – všechny úkoly jsou rozložené na dílčí podúkoly
3. Spustí soukromé sestavení aplikace (včetně testů) a ověří, že vše funguje
4. Stáhne si z úložiště nejnovější změny zdrojových kódů od ostatních členů týmu
5. Pomocí soukromého sestavení aplikace ověří, že vše stále funguje
6. Uloží své změny do úložiště
7. Integrační server provede sestavení aplikace a spuštění testů a zašle informační e-mail
8. Pokud nedojde k žádným chybám, vývojář si úkol může označit jako dokončený

## Testování

obecně o testování SW – automatizované, manuální – tady jen automatické

//zdroj?

### Jednotkové testy

### Integrační testy

selenium

<http://www.slideshare.net/sebastian_bergmann/the-php-testers-toolbox>

…

## Statická analýza kódu, kontrola kvality

Statickou analýzou zdrojového kódu označujeme proces získávání metrik kvality kódu. V kontextu kontinuální integrace získáváme vybrané metriky automatizovaně a při využití vhodných nástrojů můžeme sledovat jejich dlouhodobé trendy. [co se nedá měřit, to se nedá řídit @todo] Sledování a vyhodnocování metrik zdrojového kódu je důležité, protože nám může pomoci odhalit tzv. code smell (zatuchlý kód), který může způsobovat komplikace v pozdějších fázích softwarového projektu.

Mezi nejčastěji získávané metriky patří

* Dodržování Coding Standards (standardů pro psaní kódu)
* Množství duplicitního kódu
* komplexita a další
* pdepend, jdepend @todo

//Lepší kód -> větší zábava při programování -> spokojenější programátoři -> menší fluktuace

### Coding standards

…

Coding standards neboli Standardy pro psaní kódu označuje soubor pravidel jak formátovat zdrojový soubor…

možnost použít hotová (Java, PEAR, ZF) nebo vytvořit svoje – nedoporučuje se – noví vývojáři je nebudou znát

není poznat, kdo co psal - super, snadné pro nové vývojáře, společné vlastnictví kódu

sniffs for coding standard violations

ensures code is clean and consistent

using standard and custom coding standards

### CPD, Dead Code,

### Další code smells @todo

## Přínosy CI (//@todo Duval 64 + Ch3)

Využití kontinuální integrace může týmu i celému procesu vývoje daného software přinést různá zlepšení. V této kapitole shrnu možné přínosy využití CI podle toho, jak je rozebral [Duvall, et al., 2007 pp. 29-32].

### Snížení rizik

* chyby jsou objeveny a opraveny rychleji
  + možnost využití http://martinfowler.com/bliki/DiffDebugging.html
* lze měřit kvalitu software

### Snížení množství manuálních činností

### Nasaditelný SW je kdykoliv k dispozici

### Viditelnost projektu

### Větší důvěra týmu v kvalitu

# Výběr vhodného integračního serveru

Jak již jsem zmiňoval dříve, kontinuální integraci je možné provádět i bez využití integračního serveru, nicméně nenapadá mě žádný důvod proč. Většina serverů pro kontinuální integraci je k dispozici zdarma a jejich využití přináší další automatizaci úkonů, které by bylo jinak nutné dělat ručně. Zároveň máme jistotu, že je sestavení spouštěno vždy stejně.

## Přehled trhu serverů pro kontinuální integraci

Na trhu jsou k dispozici různé servery pro kontinuální integraci. Ten nejstarší, CruiseControl[[33]](#footnote-33), byl vyvinut ve společnosti ThoughtWorks, pro kterou pracuje Martin Fowler, autor známého článku [Fowler, 2006]. Existuje pro něj rozšíření phpUnderControl[[34]](#footnote-34), které usnadňuje prvotní nastavení CI nástrojů pro PHP, nicméně již není aktivně vyvíjeno, takže jeho nasazení bych spíše nedoporučoval. Kromě toho existují další méně známé CI servery, například Xinc[[35]](#footnote-35), který je napsaný v jazyce PHP.

Z komerčních CI serverů bych zde uvedl Bamboo[[36]](#footnote-36) od společnosti Atlassian[[37]](#footnote-37). Výhodou tohoto CI serveru je snadná integrace s dalšími nástroji pro projektové řízení od této společnosti. Dalším komerčním serverem je uBuild[[38]](#footnote-38), který je nástupcem známého serveru AnthillPro. V neposlední řadě bych uvedl integrační server zaměřený především na jazyk Java a .NET, TeamCity[[39]](#footnote-39), jehož tvůrcem je původně česká společnost JetBrains[[40]](#footnote-40). Často používané je také Team Foundation Server[[41]](#footnote-41) od společnosti Microsoft. Nejedná se jen o integrační server, ale spíše platformu zastřešující celý proces vývoje software. Je zaměřený především na vývoj ve frameworku .NET a integraci s vývojovým prostředím Visual Studio.

### Příbuzný software

S integračními servery úzce souvisí i další software, který je může doplňovat (ale funguje nezávisle na nich). Zařadil bych sem Sismo[[42]](#footnote-42), což je Continuous Testing Server – zaměřuje se jen na spouštění testů. Díky tomu ho lze využít jako komplement k CI serveru pro rychlé spouštění jednotkových testů lokálně, po commitu do repositáře ve verzovacím systému GIT. Podobně funguje i Travis CI[[43]](#footnote-43), jehož cílem je vytvořit distribuovaný systém pro spouštění testů open-source software.

Sonar[[44]](#footnote-44) je open-source nástroj na řízení kvality zdrojových kódů. V základu se zaměřuje na jazyk Java, ale pomocí open-source nebo i komerčních rozšíření je do něj možné doplnit podporu pro další jazyky (C, C#, PHP, …). Je možné ho využít ve spojení s různými servery pro kontinuální integraci jako je Hudson nebo Jenkins[[45]](#footnote-45).

## Jenkins / Hudson

Velmi známým integračním serverem je Jenkins (resp. Hudson). Někdy dochází k zaměňování těchto názvů, proto by bylo vhodné to zde upřesnit. Jeden ze zaměstnanců Sun Microsystems v roce 2006 vyvinul integrační server Hudson a uvolnil ho pod svobodnou licencí MIT. V následujících letech se vývoje účastnili i další vývojáři. V lednu 2010 společnost Oracle koupila[[46]](#footnote-46) společnost Sun Microsystems, čímž mj. získala název Hudson, který si později registrovala jako ochrannou známku. Oracle se během sjednocování infrastruktury s ostatními open-source projekty, které zastřešují, dostal do sporu[[47]](#footnote-47) s komunitou vývojářů Hudsonu a ti se rozhodli zbavit závislosti na společnosti Oracle[[48]](#footnote-48) a začali Hudson dále vyvíjet pod názvem Jenkins[[49]](#footnote-49).

Společnost Oracle nadále vyvíjí Hudson, ale komunita se spolu s hlavními vývojáři přesunula k Jenkinsu, takže rozvoj Hudsonu se zpomalil. Oba projekty jsou v současné době hostovány na Githubu, takže je možné snadno provést srovnání rychlosti vývoje a zájmu vývojářů o ně:

Tabulka : Srovnání aktivity u projektů Hudson a Jenkins na serveru GitHub.com

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Projekt** | **URL** | **Sledování** | **Forky** | **Pull Requesty** | **Commity v roce 2012** |
| Hudson | <https://github.com/hudson/hudson/> | 159 | 34 | 0 | 6 |
| Jenkins | <https://github.com/jenkinsci/jenkins/> | 1112 | 398 | 16 | 257 |

(hodnoty jsou k 25. 2. 2012)

Z těchto hodnot je vidět, že Jenkins je aktivně vyvíjen a zároveň se těší velkému zájmu uživatelů, což ho dělá vhodnější volbou než Hudson. Je to zajímavá ukázka toho, že vývoj software komunitou dobrovolníků může postupovat dopředu rychleji než vývoj software, který zaštiťuje velká společnost.

Při výuce vývoje v Javě na VŠE se dříve využíval integrační server Hudson, nicméně dnes (25. 2. 2012) již to je Jenkins[[50]](#footnote-50), což potvrzuje moji domněnku, že Jenkins je vhodnější volbou než Hudson. Vzhledem k tomu, že většina zdrojů (viz kapitola 2), které se kontinuální integrací v prostředí PHP zabývají, zvolila jako integrační server Jenkins, nenašel jsem důvod, proč vybrat jiný.

# Přehled CI nástrojů pro PHP

CI nástroje použitelné pro jazyk PHP lze rozdělit do tří velkých skupin:

* nástroje na kontrolu kvality zdrojových kódů pomocí statické analýzy
* nástroje na automatizované testování
* podpůrné nástroje pro automatizaci, tvorbu dokumentace, apod.

## PHP Lint

Kontrolu kvality kódu nemá smysl provádět dříve, než si jsme jisti, že napsaný kód je syntakticky v pořádku (to, že kód nelze zkompilovat je větší problém než nedodržování stanovených Coding Standards). Zároveň by kvůli tomu mohly nástroje pro kontrolu kvality vracet nepřesné výsledky.

Některá moderní IDE jako Zend Studio[[51]](#footnote-51), Eclipse PDT[[52]](#footnote-52) nebo Netbeans[[53]](#footnote-53) už sice kontrolu syntaktických chyb obsahují (viz Obrázek 5.1), ale slouží spíše jako upozornění pro vývojáře. Integrační nástroj nemůže spoléhat na to, že si vývojář chyby všimne a opraví ji.



Obrázek . Kontrola syntaktických chyb v editoru Zend Studio 8

Samotné PHP umožňuje provést kontrolu syntaxe PHP souboru pomocí přepínače '-l' (lint):

> php –help

-l Syntax check only (lint)

Pokud kontrolu spustíme na skriptu test1-error.php, který obsahuje syntaktickou chybu, dostaneme chybový výstup (viz Kód 5.2)

<?php

echo "Kontinuální integrace" //chybí středník

Kód . test1-error.php

> php -l test1-error.php

Parse error: syntax error, unexpected $end, expecting ',' or ';' in test1-error.php on line 2

Errors parsing test1-error.php

Kód . Chybový výstup z PHP Lint

Pokud je soubor syntakticky v pořádku (viz Kód 5.3), vrátí PHP Lint výstup (Kód 5.4)

<?php

echo "Kontinuální integrace";

Kód . Zdrojový kód souboru test2-ok.php

> php -l test2-ok.php

No syntax errors detected in test2-ok.php

Kód . Výstup z PHP Lint, když je kód v pořádku

### Kontrola dopředné kompatibility

PHP Lint můžeme použít také pro kontrolu, zda zdrojové kódy budou zkompilovatelné i v další verzi PHP. Problém může nastat například, pokud jako identifikátor použijeme slovo, které je v další verzi zařazeno mezi klíčová slova.

Pokud provedeme kontrolu pomocí PHP Lint (verze PHP 5.3) na souboru (viz Kód 5.5), tak je vše v pořádku (Kód 5.6).

<?php

class Trait {}

Kód . test3-trait.php

>php -l test3-trait.php

No syntax errors detected in test3-trait.php

Kód . Výstup z PHP Lint (verze PHP 5.3) na souboru test3-trait.php

Pokud ovšem použijeme PHP ve verzi 5.4.0alpha2, kde trait je klíčovým slovem, dostaneme chybový výstup (viz Kód 5.7)

> php -l test3-trait.php

PHP Parse error: syntax error, unexpected 'Trait' (T\_TRAIT), expecting identifier (T\_STRING) in test3-trait.php on line 2

Parse error: syntax error, unexpected 'Trait' (T\_TRAIT), expecting identifier (T\_STRING) in test3-trait.php on line 2

Errors parsing test3-trait.php

Kód . Chybový výstup z PHP Lint (verze PHP 5.4.0alpha2) na souboru test3-trait.php

Pro účely CI je důležité spouštět kontrolu syntaktické správnosti pro aktuální verzi PHP, ale může být užitečné spouštět i kontrolu, zda skripty bude možné spustit v novější verzi PHP. Vzhledem k tomu, že novější verze PHP jsou většinou rychlejší[[54]](#footnote-54), můžeme díky zajištění dopředné kompatibility snadno získat výkonové zlepšení. Podle [Bergmann, 2008] je PHP 5.3 1,2x rychlejší než PHP 5.2, takže pokud by projekt vyvíjený v době PHP 5.2 měl podobně kontrolovanou kompatibilitu s PHP 5.3, mohl jen pomocí aktualizace na novou verzi PHP získat 20% zrychlení běhu.

### Pre-commit hook

Systémy pro správu verzí jako Subversion nebo GIT umožňují navázat spouštění skriptů na události vyvolané v různých fázích ukládání souborů do úložiště. Pro kontrolu syntaktické správnosti souborů ještě před jejich přijetím do úložiště využijeme pre-commit hook, který může probíhající commit zrušit. Lze využít už hotové skripty pro SVN[[55]](#footnote-55) nebo GIT[[56]](#footnote-56). Oba fungují tak, že získají seznam změněných souborů s příponou PHP, spustí na nich kontrolu syntaktické správnosti a pokud jsou všechny soubory v pořádku, tak commit umožní dokončit. V opačném případě vrátí textovou informaci, které soubory obsahují syntaktické chyby.

PHP kód nemusí být ovšem uložený jen v souborech s příponou PHP. Například často používaný PHP framework Zend Framework[[57]](#footnote-57) využívá jako šablonovací jazyk přímo PHP a šablony jsou ve výchozím nastavení uloženy v souborech s příponou PHTML (viz Kód 5.8). Proto by bylo vhodné, aby se během commitu kontrolovala i jejich syntaktická správnost, což výše uvedené skripty neřeší.

<h1><?php echo "Kontinuální integrace"; ?></h1>

Kód . Příklad šablony v PHTML souboru (test4-template.phtml)

## PHP\_CodeSniffer

PHP\_CodeSniffer[[58]](#footnote-58) je nástroj určený na analýzu zdrojových kódů aplikace a kontrolu, zda odpovídají zvoleným standardům pro formátování souborů se zdrojovým kódem.

Je možné zvolit již nějaký předdefinovaný standard pro formátování (k dispozici jsou například standardy používané v knihovně PEAR, Zend Frameworku a dalších), případně je možné si vytvořit pravidla na míru pro svůj projekt.

Pro aplikace postavené nad Zend Frameworkem je samozřejmostí zvolit předdefinovaný standard. Nicméně provedl jsem jednu úpravu a upravil jsem pravidlo na omezení délky řádků zdrojového kódu. Výchozích 80 znaků je podle mého názoru přežitek z doby, kdy byla omezením šířka terminálu. Dnes, kdy je běžné vyvíjet na monitorech s velkým rozlišením (fullHD – 1920x1080 bodů) nemá smysl omezovat délku řádku na méně než 120 znaků.

@todo ukázkový skript a demo výstup

PHP\_CodeSniffer je možné podobně jako PHP Lint spouštět jako pre-commit hook. Nicméně při větším množství změněných souborů v rámci jednoho commitu může kontrola trvat dlouho, takže doporučuji provádět kontrolu dodržování coding standards až v rámci kontinuální integrace.

@todo – jak umí kontrolovat JS/CSS?

## PHP CPD

PHP Copy/Paste Detector[[59]](#footnote-59) je nástroj, který odhalí zkopírované nebo duplicitní bloky kódu. Jeho použití je jednoduché (viz Kód 5.9).

> phpcpd .

phpcpd 1.3.2 by Sebastian Bergmann.

Found 1 exact clones with 24 duplicated lines in 1 files:

- ArticlesController.php:14-38

ArticlesController.php:67-91

20.34% duplicated lines out of 118 total lines of code.

Kód . Ukázka použití PHP CPD

Samozřejmostí je výstup do XML použitelného pro CI server, který lze aktivovat přepínačem ‑‑log-pmd <file>.

## PHP Depend

<http://pdepend.org/>

statická analýza

- závislosti tříd,

- vysvětlit pyramidku

<http://pdepend.org/documentation/software-metrics.html>

<http://pdepend.org/documentation/handbook/reports/overview-pyramid.html#ml06>

<http://pdepend.org/documentation/handbook/reports/abstraction-instability-chart.html>

pdepend --jdepend-chart=jdepend.svg --overview-pyramid=pyramid.svg --summary-xml=sum.xml ./models

## PHP MD

PHP Mess Detector[[60]](#footnote-60) je nástroj na analýzu zdrojových kódů v PHP a detekci potenciálně problematických míst. Snaží se být pro PHP podobným nástrojem, jako je PMD[[61]](#footnote-61) pro Javu.

- kvalita kódu

- krátké proměnné – ignoruje v cyklech apod.

pear channel-discover pear.phpmd.org

pear channel-discover pear.pdepend.org

pear install --alldeps phpmd/PHP\_PMD

## Neinicializované proměnné

<http://code.google.com/p/php-initialized/>

?budu zařazovat? – zkusit jak moc rozbíjí Shopio

## PHPLOC

PHPLOC[[62]](#footnote-62) je jednoduchý nástroj, který umožňuje získat přehled o počtu řádků kódu v projektu a počítá některé další metriky (@todo rozpracovat v teoretické kapitole). Jeho použití je jednoduché (viz Kód 5.10).

> phploc ./application

phploc 1.6.1 by Sebastian Bergmann.

Directories: 59

Files: 290

Lines of Code (LOC): 44338

Cyclomatic Complexity / Lines of Code: 0.05

Comment Lines of Code (CLOC): 8176

Non-Comment Lines of Code (NCLOC): 36162

Namespaces: 0

Interfaces: 0

Classes: 290

Abstract: 6 (2.07%)

Concrete: 284 (97.93%)

Average Class Length (NCLOC): 129

Methods: 1182

Scope:

Non-Static: 1164 (98.48%)

Static: 18 (1.52%)

Visibility:

Public: 796 (67.34%)

Non-Public: 386 (32.66%)

Average Method Length (NCLOC): 31

Cyclomatic Complexity / Number of Methods: 2.48

Anonymous Functions: 0

Functions: 10

Constants: 280

Global constants: 2

Class constants: 278

Kód .10 Ukázkový výstup z aplikace phploc

Samozřejmostí je opět výstup do formátu použitelného pro CI server, který lze aktivovat přepínačem ‑‑log-csv <file>.

## Automatizované testování

PHPUnit

@todo selenium

## Generování API dokumentace

Při psaní zdrojového kódu je běžné do něj doplňovat tzv. dokumentační komentáře. Lze z nich totiž pak vygenerovat programátorskou dokumentaci (také nazývanou API dokumentace), kterou lze procházet snadněji než zdrojový kód.

Způsob zápisu používaný v jazyce PHP je založený na syntaxi používané v jazyce Java a nástroji Javadoc[[63]](#footnote-63), jen je upravený kvůli odlišnostem jazyka PHP.

<?php

/\*\*

\* Ukázková třída

\*

\* @author Martin Hujer

\*/

class MyClass

{

/\*\*

\* Funkce foo nic nedělá, jen vrátí předaný parametr.

\*

\* @param boolean $param Předaný parametr

\* @return boolean

\*/

public function foo($param)

{

return $param;

}  
}

Kód .11: Ukázka možnosti využití PHPDoc (soubor phpdoc01.php)

### PhpDocumentor

PhpDocumentor[[64]](#footnote-64) je původní nástroj na generování API dokumentace pro PHP skripty. Ovládá se pomocí příkazové řádky, takže ho není problém zařadit do skriptu pro sestavení. Jeho spuštění může vypadat například takto:

phpdoc -f phpdoc01.php -t doc

Obrázek 2 naznačuje, jak vypadá výsledná dokumentace ve výchozí HTML šabloně.



Obrázek : Ukázka dokumentace vygenerované pomocí skriptu PhpDocumentor

Nevýhodou PhpDocumentor je nepodpora[[65]](#footnote-65) kódování UTF-8. Nicméně je možné ji doplnit úpravou[[66]](#footnote-66) šablon pro generování dokumentace.

### Docblox

Docblox[[67]](#footnote-67) je nový nástroj na generování API dokumentace. Jednou z jeho předností je rychlost generování. To je dobře vidět například na dokumentaci Zend Frameworku, která se s využitím nástroje PhpDocumentor generovala 100 minut, tak DocBlox ji zvládne vygenerovat za přibližně 10 minut[[68]](#footnote-68). Další zajímavou funkcí je generování schématu tříd ve formátu SVG.

Generování dokumentace v DocBloxu spustíme velmi podobně jako v případě PhpDocumentoru:

docblox -f phpdoc01.php -t docblox

Jak vypadá vygenerovaná dokumentace ukazuje Obrázek 3.



Obrázek : Ukázka dokumentace vygenerované nástrojem DocBlox

#### Validace dokumentačních komentářů

Docblox při generování dokumentace zároveň kontroluje, zda dokumentační komentáře u zdrojového kódu obsahují potřebné informace. Případné nedostatky standardně reportuje do sekce v HTML dokumentaci. Pro snadnější napojení na kontinuální integraci existuje šablona „checkstyle“[[69]](#footnote-69), která přehled problematických míst vygeneruje do XML souboru.

Šablonu je možné nainstalovat přímo přes Docblox:

docblox template:install checkstyle

Upravený příkaz pro kontrolu dokumentace vypadá takto:

docblox -f phpdoc01.php -t docblox --template checkstyle

### ApiGen

ApiGen[[70]](#footnote-70) je další nový nástroj na generování dokumentace ze zdrojových kódů v jazyce PHP. Je zajímavý například tím, že podporoval generování dokumentace pro PHP 5.4 už dříve, než bylo vydáno, a také tím, že je vyvíjen v České republice.

Spustíme ho opět snadno z příkazové řádky:

apigen -s phpdoc01.php -d apigen

Jak vypadá vygenerovaná dokumentace ukazuje Obrázek 4.



Obrázek : Ukázka dokumentace vygenerovaná nástrojem ApiGen

#### Validace dokumentačních komentářů

ApiGen stejně jako DocBlox provádí kontrolu, zda jsou dokumentační komentáře vyplněny a případné chyby také umožňuje exportovat do formátu pro Checkstyle:

apigen -s phpdoc01.php -d apigen --report apigen/checkstyle.xml

### Výběr nástroje pro generování dokumentace

Pro volbě nástroje pro generování dokumentace ze zdrojových kódů máme na výběr z několika možností. Vzhledem k tomu, že ostatní součásti kontinuální integrace na vygenerované dokumentaci nezávisí, je možné nástroj snadno vyměnit i později.

Nakonec jsem pro využití v kontinuální integraci zvolil nástroj ApiGen, protože má subjektivně přehlednější dokumentaci než DocBlox. PhpDocumentor jsem nevybral, protože neumožňuje vytvoření reportu pro Checkstyle.

## Phing

@todo

Phing / Ant + další build nástroje –

Phing – obsahuje hotové tasky pro PHP tooly #win

//@todo Phing - povídání o targetech a podobně; XML, Oxygen + schéma

//@todo Povídání o XML - spíš vynechám, že top přesahuje rozsah této práce

## HipHop

hiphop analysis - měl by dělat i validaci PHP, lepší než samotné php :)

http://sebastian-bergmann.de/archives/894-Using-HipHop-for-Static-Analysis.html

## JS lint?

Možná je nějaký modul v PHP CS

http://zdrojak.root.cz/clanky/kontrola-javascriptu-s-jslint-a-jshint/

http://www.tomas-dvorak.cz/clanky/jshint-a-spousteni-validace-javascriptu-z-prikazove-radky

CI crawl & check for 200 - místo testování, jen zběžná kontrola, jestli se něco nerozbilo

# Instalace CI platformy pro malou firmu

V této kapitole se zaměřím na postup instalace CI platformy pro malou firmu, nebudu se tedy zabývat tématy, která se netýkají kontinuální integrace v malé firmě, jako je koordinace spolupráce více integračních serverů, distribuované spouštění testů a další.

## Skript pro sestavení aplikace

Vývoj a testování skriptu pro sestavení je vhodné provádět spíše na menším projektu než na rozsáhlé aplikaci. Důvodem je to, že pokud bychom vybrali rozsáhlou aplikaci se stovkami PHP souborů, tak bychom museli čekat zbytečně dlouhou dobu na proběhnutí jednotlivých částí sestavení.

Proto jsem pro vytvoření skriptu pro sestavení aplikace zvolil ukázkovou aplikaci ZFTutorial, která je výsledkem návodu Getting Started with Zend Framework[[71]](#footnote-71), pro úvod do funkčnosti Zend Frameworku[[72]](#footnote-72).

Protože aplikace je k dispozici jen jako archív ve formátu ZIP a pro kontinuální integraci je důležité využívat verzovací systém, vytvořil jsem GIT repositář[[73]](#footnote-73) na serveru GitHub.com a aplikaci tam uložil.

@todo dopíšu tam nějaké testy?

@todo Shopio budu integrovat jen interně a budu z něj dělat ukázky metrik

Jako první krok tvorby skriptu pro sestavení aplikace jsem si v kořenovém adresáři projektu vytvoříme prázdný soubor build.xml. Do něj vložíme kostru projektu a jednoduchý „task“ echo, abychom ověřili, že Phing funguje správně.

<project name="zf-tutorial" default="main">

<target name="main">

<echo message="Phing works!"/>

</target>

</project>

Po zavolání příkazu phing v adresáři, kde je uložen soubor build.xml bychom měli dostat tento výstup:

> phing

Buildfile: I:\\_BP\zf-tutorial\build.xml

zf-tutorial > main:

[echo] Phing works!

BUILD FINISHED

Total time: 0.3852 seconds

## PHP Lint

Prvním krokem, který je nutné při sestavení udělat je kontrola syntaktické správnosti, pro jazyk PHP je to tzv. PHP Lint (viz kapitola 5.1).

Do build skriptu přidáme nový target:

<target name="lint">

<phplint haltonfailure="true">

<fileset dir="${project.basedir}/application">

<include name="\*\*/\*.php"/>

<include name="\*\*/\*.phtml"/>

</fileset>

</phplint>

</target>

Je zde několik věcí, které je potřeba vysvětlit. Phing již v základu obsahuje task phplint[[74]](#footnote-74), který na pozadí spouští samotnou kontrolu pomocí php -l.

Atribut haltonfailure="true" znamená, že pokud selže syntaktická kontrola jakéhokoliv souboru, tak ihned selže celý build a nebudou se provádět další kroky (například analýza coding standards).

Proměnná ${project.basedir} se automaticky nastaví na adresář, který je nadřazený souboru build.xml.

Fileset popisuje seznam souborů, která se předají tasku (akci?). Bylo by možné stejný fileset zkopírovat do dalších částí buildskriptu, protože na zdrojových kódech samotné aplikace budeme spouštět i další tasky. Nicméně Phing toto umožňuje řešit elegantněji. Je možné vytvořit fileset i uvnitř tagu <project>, přiřadit mi identifikátor a na něj pak jen odkazovat. (@todo DRY).

Fileset si označíme jako "src":

<fileset id="src" dir="${project.basedir}/application">

<include name="\*\*/\*.php"/>

<include name="\*\*/\*.phtml"/>

</fileset>

A target pak můžeme zpřehlednit takto:

<target name="lint">

<phplint haltonfailure="true">

<fileset refid="src"/>

</phplint>

</target>

Podobně jsem si v buildskriptu nadefinoval adresáře tests a library a doplnil je do tasku phplint.

Jak vypadá výsledný buildskript se můžete podívat do {@todo příloha 1}.

## Instalace serveru Jenkins

//@ nezávislé na platformě

Pro ukázkovou instalaci CI serveru jsem zvolil aktuální verzi serveru Jenkins (1.454). Tyto vycházejí velmi často (většinou každý týden), což nemusí být vhodné pro instalace, které musí být stabilní. Nasazení nové verze s sebou přináší nutnost otestovat, zda skript pro sestavení a veškerá rozšíření fungují jako dříve. Jenkins proto zavedl tzv. LTS verze (Long-Term Support)[[75]](#footnote-75), které jsou založeny na některé starší verzi, která se osvědčila a jsou do nich backportovány jen opravy chyb a ne nová funkcionalita.

Jenkins je k dispozici jak ve formátu Java Web Archive (.war), tak i jako instační balíčky pro operační systém Windows, a běžné linuxové distribuce. Pro některé linuxové distribuce jsou k dispozici i úložiště balíčků[[76]](#footnote-76), takže je možné Jenkins snadno aktualizovat na novou verzi.

Instalaci začneme stažením balíčku ve formátu .war, který umístíme do vybrané složky (v mém případě I:/BP-jenkins/). Jenkins si ve výchozím stavu ukládá nastavení a soubory do domovského adresáře aktuálně přihlášeného uživatele. Vhodnější je si data ukládat do nějakého jiného adresáře, což lze nastavit do hodnoty proměnné prostředí JENKINS\_HOME. Samotný Jenkins se poté spustí jako běžná aplikace v Javě pomocí příkazu java -jar jenkins.war.

Než zadávat tyto příkazy při každém spouštění Jenkinsu ručně, je lepší si vytvořit spouštěcí dávkový soubor. Pojmenujeme ho například jenkins-start.cmd a obsah bude tento:

set JENKINS\_HOME=i:\BP-jenkins\data\

java -jar jenkins.war

Po jeho spuštění začne startovat Jenkins. Z výpisu je zřejmé, že se použil námi určený adresář:

Jenkins home directory: i:\BP-jenkins\data found at: EnvVars.masterEnvVars.get("JENKINS\_HOME")

Posledním řádkem výpisu by měla být informace oznamující, že spuštění proběhlo v pořádku:

INFO: Jenkins is fully up and running

Ověříme to otevřením URL http://localhost:8080/ v prohlížeči. Měli bychom vidět hlavní stránku serveru Jenkins (viz Obrázek 6). Tím máme hotovou základní instalaci a můžeme se pustit do vytvoření projektu.



Obrázek : Hlavní stránka Jenkinsu po prvním spuštění

## Možnosti konfigurace

Jedna z možností, jak ovládat a konfigurovat Jenkins je přes webové rozhraní, jak jsme viděli výše. Další možností je pomocí klienta příkazové řádky. Jeho výhodou je možnost snadno spustit více příkazů za sebou, bez nutnosti manuálního procházení webových stránek. Nevýhodou je nemožnost ovládat vše, co je možné z webového rozhraní. Potřebnou verzi CLI klienta lze snadno získat přímo z našeho Jenkins serveru pomocí příkazu:

wget http://localhost:8080/jnlpJars/jenkins-cli.jar

Možná jste si všimli, že se Jenkins automaticky načetl v jazyce, který máte nastavený jako výchozí ve webovém prohlížeči. S největší pravděpodobností to tedy byla čeština. Český překlad Jenkinsu není kompletní, což by znepříjemňovalo práci a zároveň mohlo být v některých situacích matoucí. Lze to vyřešit buď nastavením prohlížeče, aby požadoval webové stránky v angličtině, což by museli provést všichni, kteří budou Jenkins využívat. Vhodnějším řešením je instalace rozšíření Locale[[77]](#footnote-77), které umožňuje nastavit jazyk napevno, bez ohledu na konfiguraci webového prohlížeče. Rozšíření nainstalujeme takto:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin locale

Server je po instalaci rozšíření vždy nutné restartovat, což lze provést příkazem:

java -jar jenkins-cli.jar -s http://localhost:8080 safe-restart

(Nicméně pokud je server spuštěný je pomocí java -jar jenkins.war, tak restart nelze provést a je potřeba server ukončit a poté opět spustit.)

Teď už je rozšíření aktivní, takže můžeme přejít do nastavení systému[[78]](#footnote-78) a sekci Locale upravit takto:



Obrázek : Jenkins - nastavení anglického prostředí

Z historických důvodů Hudson a Jenkins používají pro označení úspěšných sestavení modrou barvu (resp. modré kuličky), což nemusí být na první pohled zřejmé (uživatelé spíše čekají zelenou barvu). Lze to vyřešit instalací rozšíření Green Balls[[79]](#footnote-79).

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin greenballs

(server opět restartujeme)

## Vytvoření projektu

Již je vše připravené a můžeme se pustit do přípravy automatizovaného sestavování. Budeme potřebovat dvě rozšíření - GIT[[80]](#footnote-80) a Phing[[81]](#footnote-81).

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin phing

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin git

@todo instalace gitu přesahuje rozsah této práce, předpokládám, že je v PATH

@todo instalace phingu - přes pear – viz výše

Na úvodní stránce zvolíme možnost „New Job“ („Job“ je označení, které Jenkins používá pro něco, co chápeme jako „projekt“), vyplníme jméno (v našem případě zf-tutorial), dále vybereme volbu „Build a free-style software project“.

V dalším kroku v sekci „Source Code Management“ vložíme URL repositáře s naším projektem (https://mhujer@github.com/mhujer/zf-tutorial.git) a do pole „Branches to build“ zadáme „master“ (chceme sestavení provádět jen na hlavní větvi projektu).

V sekci „Build“ zvolíme „Add build step“ a „Invoke Phing targets“. Nic dalšího není nutné nastavovat, protože soubor s instrukcemi pro sestavení máme pojmenovaný standardně, a máme i nastavený výchozí „target“. Klikneme tedy dole na „Save“ a poté vlevo v nabídce na „Build Now“.

Vidíme, že sestavení proběhlo v pořádku.



Obrázek : Úspěšné sestavení v Jenkinsu

Do detailních informací o sestavení se dostaneme kliknutím na datum a čas konkrétního sestavení v části „Build History“. V sekci „Console Output“ vidíme, průběh sestavení (pokud by build trval déle, tak tu lze vidět aktuální průběh).

Dále je vidět, že se spustil target main, který jen vypsal informační hlášku.

zf-tutorial > main:

[echo] Phing works!

Přejdeme proto do nastavení projektu a sekci „Build“ vyplníme do pole „Targets“ „lint“, což je název targetu, který jsme si vytvořili v kapitole XY (@todo). Spustíme další sestavení pomocí „Build Now“ (@todo proč ne automaticky teď?).

V konzoli teď vidíme, že už se build spustil tak, jak měl:

zf-tutorial > lint:

[phplint] I:\BP-jenkins\data\jobs\zf-tutorial\workspace\application\Bootstrap.php: No syntax errors detected

.........

[phplint] I:\BP-jenkins\data\jobs\zf-tutorial\workspace\tests\library\bootstrap.php: No syntax errors detected

BUILD FINISHED

Total time: 1.0731 second

Finished: SUCCESS

## Nasazení PHP\_CodeSniffer

PHP\_CodeSniffer lze nainstalovat pomocí PEAR následujícím příkazem:

pear install PHP\_CodeSniffer

@todo popsat pear

Do skriptu pro sestavení jsem přidal kontrolu Coding Standards s použitím standardu pro Zend Framework a exportem chyb ve formátu pro Checkstyle.

<target name="phpcs" depends="prepare" description="Kontrola Coding standards">

<phpcodesniffer standard="Zend">

<fileset refid="src"/>

<fileset refid="tests"/>

<fileset refid="library"/>

<formatter type="default" usefile="false"/>

<formatter type="checkstyle" outfile="${project.basedir}/build/checkstyle-phpcs.xml"/>

</phpcodesniffer>

</target>

Dále jsem přidal další target, v tomto případě nebude vykonávat žádnou činnost, ale bude záviset na ostatních, takže zajistí, že ty se spustí dříve.

<target name="build" depends="prepare, lint, phpcs" description="Meta target, spouští ostatní targety"/>

Vzhledem k tomu, že PHP\_CodeSniffer generuje tzv. artefakt (soubor, který je výsledkem buildu), je nutné při dalších spuštěních zajistit, aby tento soubor neexistoval a vygeneroval se znovu. Toho lze dosáhnout pomocí dalších targetů, jeden pro smazání adresáře a druhý pro jeho znovuvytvoření. Je vhodné upravit všechny ostatní targety, aby závisely na targetu prepare, neboť je pak bude možné spouštět i samostatně.

<target name="cleanup" description="Vyčistění workspace">

<delete dir="${project.basedir}/build"/>

</target>

<target name="prepare" depends="cleanup" description="Příprava workspace">

<mkdir dir="${project.basedir}/build"/>

</target>

Zároveň bylo nutné upravit zdrojový kód ukázkové aplikace, aby tam existovalo nějaké porušení coding standards.

Do Jenkinse je nutné doinstalovat rozšíření Checkstyle, které umí zpracovat XML soubor se zjištěnými chybami:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin checkstyle

Po restartu serveru je dále nutné v editaci projektu v sekci Post-build Actions vybrat volbu Publish Checkstyle analysis results a do pole Checkstyle results vyplnit build/checkstyle-phpcs.xml. Jenkins vypíše chybové hlášení, že soubor neexistuje, což je správně, protože bude vygenerován až při dalším sestavení.

Posledním krokem je odstranění názvu vybraného targetu v sekci Build (použije se tedy výchozí nastavený v skriptu pro sestavení).

## Nasazení ApiGen

Dalším krokem je přidání generování dokumentace pomocí nástroje ApiGen popsaného v kapitole 5.9.3.

Nejprve je nutné přidat target pro ApiGen (je využit přímo task ApiGen, který už je součástí testovací verze nástroje Phing, nicméně není součástí existující stabilní verze 2.4.9):

<target name="apigen" depends="prepare" description="Generování dokumentace">

<apigen source="${project.basedir}/application" destination="${project.basedir}/build/docs" report="${project.basedir}/build/checkstyle-apigen.xml"/>

</target>

Jenkins je schopen na hlavní stránku projektu doplnit odkazy na další HTML výstupy, toho lze dosáhnout rozšířením HTML Publisher[[82]](#footnote-82), které nainstalujeme pomocí:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin htmlpublisher

Po restartu je třeba HTML Publisher nakonfigurovat. V editaci nastavení projektu v sekci Post-build Actions musí být zaškrtnuté pole Publish HTML reports. Adresář s daty (HTML directory to archive) je build/docs. Report title je vhodné nastavit například na „API Docs“. Ostatní volby je možné ponechat ve výchozím nastavení.

ApiGen generuje i soubor s chybami v dokumentačních komentářích, pročež je nutné upravit pole „Checkstyle results“ na build/checkstyle-\*.xml (hvězdičkou dosáhneme toho, že se budou načítat všechny XML soubory z daného adresáře začínající na checkstyle-.

## Nasazení PHPCPD

Nástroj na kontrolu duplicitního kódu PHPCPD je popsán v kapitole 5.3. Do systému ho lze nainstalovat pomocí příkazu

pear install phpunit/phpcpd

Dále je nutné ho zařadit do skriptu pro sestavení. Opět je možné využít již předpřipravený task v rámci nástroje Phing:

<target name="phpcpd" depends="prepare">

<phpcpd>

<fileset refid="src"/>

<fileset refid="tests"/>

<fileset refid="library"/>

<formatter type="pmd" outfile="${project.basedir}/build/pmd-cpd.xml"/>

</phpcpd>

</target>

Aby bylo možné ověřit, že nástroj kontroluje kód správně, je vhodné například zduplikovat nějaký PHP soubor v testovací aplikaci (ale bude nutné přejmenovat třídu v něm, aby nedošlo k problémům při generování dokumentace.

Pro Jenkins existuje přímo rozšíření, které umí zpracovat výsledky analýzy duplicitního kódu, DRY (zkratka pro Don't Repeat Yourself). Instalaci lze provést pomocí příkazu:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin dry

Zároveň je nutné nastavit projekt v Jenkinsu. Zaškrtnutím pole Publish duplicate code analysis results v sekci Post-build Actions se aktivují další pole. Do Duplicate code results vložíme cestu k souboru, který nám PHPCPD generuje: build/pmd-cpd.xml.

Po provedení sestavení máme k dispozici i informace o duplicitním kódu.

## Nasazení PHPLOC

PHPLOC je nástroj, který počítá různé metriky zdrojových kódů a výsledky umí exportovat do CSV souboru využitelného v Jenkinsu. Detailně byl popsán v kapitole 5.7.

Instalaci lze provést pomocí PEARu: (@todo nepřesunu instalace všeho do teoretické kapitoly a tady jen předpokládat, že jsou nainstalované?)

pear install phpunit/phploc

Do skriptu pro sestavení se spouštění PHPLOC doplní přidáním další targetu:

<target name="phploc" depends="prepare" description="Analýza PHPLOC">

<exec command="phploc --log-csv ${project.basedir}/build/phploc.csv ${project.basedir}/application" logoutput="true" />

</target>

Do Jenkinse je nutné nainstalovat rozšíření Plot[[83]](#footnote-83), které umí vykreslovat grafy z hodnot získaných v jednotlivých sestaveních.

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin plot

@todo Popis jeho detailního nastavení přesahuje rozsah této práce (a nebo to radši rozepíšu?).

Nastavení je proto možné buď vytvořit ručně podle tabulek;

XML: je připraveno v samostatném XML souboru https://github.com/mhujer/bakalarka/blob/master/jenkins/plot.xml. Ten do Jenkinse n@todo

## Nasazení PDepend

PDepend je nástroj umožňující získávat různé metriky zdrojových kódů. Jeho detailní popis je možné nalézt v kapitole 5.4.

Do skriptu pro sestavení ho lze přidat opět pomocí předpřipraveného tasku v nástroji Phing. Dostal opět jen soubory, které jsou přímo součástí aplikace, nebudeme metriky získávat pro použité knihovny apod. PDepend generuje jak data pro rozšíření JDepend, tak i dvě přehledová schémata.

<target name="pdepend" depends="prepare" description="Analýza nástrojem PDepend">

<phpdepend>

<fileset refid="src"/>

<logger type="jdepend-xml" outfile="${project.basedir}/build/jdepend.xml"/>

<logger type="jdepend-chart" outfile="${project.basedir}/build/dependencies.svg"/>

<logger type="overview-pyramid" outfile="${project.basedir}/build/overview-pyramid.svg"/>

</phpdepend>

</target>

Do Jenkinse je nutné nainstalovat rozšíření JDepend[[84]](#footnote-84):

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin jdepend

Po restartu serveru je nutné mu nastavit cestu k souboru, který se generuje během sestavení. V sekci Post-build Actions je nutné zaškrtnout pole Report JDepend a do pole Pre-generated JDepend File zadat cestu: build/jdepend.xml.

Na zobrazení vygenerovaných schémat není nutné do Jenkinse instalovat žádné rozšíření. Vzhledem k tomu, že jsou to obrázky ve formátu SVG, které umí dnešní běžně využívané prohlížeče bez problémů zobrazit, tak stačí na hlavní stránce projektu kliknout na odkaz „add description“ a do textového pole vložit tento kód:

<img src="http://localhost:8080/job/zf-tutorial/ws/build/dependencies.svg"/>

<img src="http://localhost:8080/job/zf-tutorial/ws/build/overview-pyramid.svg"/>

Schémata se poté budou zobrazovat na hlavní stránce.

## Nasazení PHPMD

PHPMD (Mess Detector) je nástroj na odhalení potenciálních problémů ve zdrojových kódech. Je závislý na nástroji PDepend, jehož výstup používá jako vstup pro vlastní analýzu.

Je vhodné vytvořit si vlastní XML soubor s pravidly[[85]](#footnote-85), která bude PHPMD ověřovat. Výhodou tohoto přístupu je, že pokud v budoucnu bude potřeba některá pravidla doplnit nebo upravit, nebude nutné upravovat skript pro sestavení. Soubor pravidel je uložený v souboru phpmd.xml v kořenovém adresáři projektu.

Skript pro sestavení je nutné doplnit o následujcí kód:

<target name="phpmd" depends="prepare" description="PMD analýza">

<phpmd rulesets="${project.basedir}/phpmd.xml">

<fileset refid="src"/>

<formatter type="xml" outfile="${project.basedir}/build/pmd.xml"/>

</phpmd>

</target>

Podporu pro zpracování výstupů z PHPMD lze do Jenkinse doplnit pomocí příkazu:

java -jar jenkins-cli.jar -s http://localhost:8080 install-plugin pmd

Po restartu je nutné nastavit načítání výstupů v našem projektu. V sekci „Publish PMD analysis results“ vložíme do pole „PMD results“ cestu k souboru build/pmd.xml.

## Úvod - automatizace sestavení

Jak již jsem popisoval v teoretické části, kontinuální integrace vyžaduje automatizovaný build. Integrační server Jenkins, který jsem zvolil pro implementaci, podporuje různé nástroje na automatizaci sestavení – jako například Apache Ant, Phing, Rake a další @todo (+linky). Pro účely této práce jsem zvolil Phing, který je napsaný v jazyce PHP a umožnuje @todo vlastní classy @todo výběr možná přehodím do teorie

## Výběr operačního systému

Jako operační systém pro instalaci CI platformy jsem zvolil Linux, konkrétně distribuci Debian (http://www.debian.org/). Instalace samotného operačního systému překračuje rozsah této práce, proto v dalších krocích budu předpokládat nainstalovaný základní verzi operačního systému Debian ve verzi 6.

apt-get update

apt-get upgrade -y

apt-get install htop -y

apt-get install php5 -y

apt-get install php-pear -y

## Instalace balíčků přes PEAR

CI server řeší automatické spouštění buildu 🡪 nejdříve potřebuji build skript v Phingu, který se bude dát spouštět lokální (multiplatformní - Win / Lin)

* nastavení od A-Z
* Úprava procesů
* praktické použití na nějakém projektu
* //integrační problémy při použití branchí

-- možná ukazovat na ZF quickstartu?

- "artefakty"

- php project wizard https://github.com/sebastianbergmann/php-project-wizard

- <http://jenkins-php.org/>

<https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+and+PHP>

# Praktická implementace CI platformy v malé firmě

procesy a spol

# Závěr

* Má Zend Framework Continous Integration server?
* <http://www.youbrokethebuild.com/>
* Continuous Integration anti-patterns <http://www.ibm.com/developerworks/java/library/j-ap11297/>¨

Automatizace automatizace

Celé by to šlo vzít z https://github.com/sebastianbergmann/php-jenkins-template/blob/master/config.xml ale je naprd mít blackbox, který nechápeme.

Zbuildovat Shopio po revizích – alespoň 100

Překlad Jenkinse? Ať si ustanovím terminologii ☺

# Terminologický slovník

|  |  |  |
| --- | --- | --- |
| commit |  |  |
| build | sestavení |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Seznam použité literatury a zdrojů

**Beck, Kent. 2005.** *Extreme Programming Explained: Embrace Change.* Boston, MA : Addison-Wesley, 2005. ISBN: 0321278658.

**—. 2002.** *Test-driven Development: By Example.* Boston : Addison-Wesley Professional, 2002. ISBN: 0321146530.

**Bergmann, Sebastian. 2008.** Benchmark of PHP Branches 3.0 through 5.3-CVS. [Online] 7. Únor 2008. [Citace: 20. Červenec 2011.] http://sebastian-bergmann.de/archives/745-Benchmark-of-PHP-Branches-3.0-through-5.3-CVS.html.

**—. 2011.** *Integrating PHP Projects with Jenkins.* Sebastopol : O'Reilly Media, Inc., 2011. 978-1-4493-0943-5.

**Duvall, Paul, Matyas, Stephen and Glover, Andrew. 2007.** *Continuous Integration: Improving Software Quality and Reducing Risk.* Upper Saddle River, NJ : Addison-Wesley Professional, 2007. http://www.amazon.com/Continuous-Integration-Improving-Software-Reducing/dp/0321336380. ISBN: 978-0321336385.

**Fowler, Martin. 2006.** Continuous Integration. [Online] 1. Květen 2006. [Citace: 18. Červenec 2011.] http://martinfowler.com/articles/continuousIntegration.html.

**Humble, Jez a Farley, David. 2010.** *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Upper Saddle River, NJ : Addison-Wesley, 2010. ISBN: 9780321601919.

**McConnell, Steve. 2006.** *Odhadování softwarových projektů.* Brno : Computer Press, 2006. 80-251-1240-3.

**Menšík, Vlastimil. 2009.** CI za použití Hudsonu a Mavenu. *ET NETERA.* [Online] ET NETERA, 20. říjen 2009. [Citace: 27. červenec 2011.] http://www.etnetera.cz/cz/21447-tech\_streda/ci\_za\_pouziti\_hudsonu\_a\_mavenu.html.

**Smart, John Ferguson. 2011.** *Jenkins: The Definitive Guide.* Sebastopol : O'Reilly Media, 2011. 978-1-4493-8959-8.

**van Dam, Michelangelo. 2011.** Improving QA on PHP development projects. *Zend.com.* [Online] Zend Technologies Ltd., 14. Duben 2011. [Citace: 29. Únor 2012.] http://www.zend.com/en/resources/webinars/#QAPHP.

**Wikipedia contributors. 2011.** Continuous integration. *Wikipedia, The Free Encyclopedia.* [Online] Wikipedia Foundation, 20. červenec 2011. [Citace: 27. červenec 2011.] http://en.wikipedia.org/w/index.php?title=Continuous\_integration&oldid=440518476.

**Zikmund, Štěpán. 2011.** Diplomová práce: Deployment aplikací v PHP. Praha : KIT FIS VŠE, 2011.

# Seznam obrázků, tabulek a ukázek kódu

## Seznam obrázků

[Obrázek 1.1 Schéma vodopádového modelu vývoje software 8](file:///I:\_BP\BP.docx#_Toc318898220)

[Obrázek 5.1 Kontrola syntaktických chyb v editoru Zend Studio 8 19](#_Toc318898221)

[Obrázek 2: Ukázka dokumentace vygenerované pomocí skriptu PhpDocumentor 25](#_Toc318898222)

[Obrázek 3: Ukázka dokumentace vygenerované nástrojem DocBlox 26](#_Toc318898223)

[Obrázek 4: Ukázka dokumentace vygenerovaná nástrojem ApiGen 27](#_Toc318898224)

## Seznam tabulek

[Tabulka 1: Srovnání aktivity u projektů Hudson a Jenkins na serveru GitHub.com 18](#_Toc318898225)

## Seznam ukázek kódu

[Kód 5.1 test1-error.php 19](#_Toc318898226)

[Kód 5.2 Chybový výstup z PHP Lint 19](#_Toc318898227)

[Kód 5.3 Zdrojový kód souboru test2-ok.php 20](#_Toc318898228)

[Kód 5.4 Výstup z PHP Lint, když je kód v pořádku 20](#_Toc318898229)

[Kód 5.5 test3-trait.php 20](#_Toc318898230)

[Kód 5.6 Výstup z PHP Lint (verze PHP 5.3) na souboru test3-trait.php 20](#_Toc318898231)

[Kód 5.7 Chybový výstup z PHP Lint (verze PHP 5.4.0alpha2) na souboru test3-trait.php 20](#_Toc318898232)

[Kód 5.8 Příklad šablony v PHTML souboru (test4-template.phtml) 21](#_Toc318898233)

[Kód 5.9 Ukázka použití PHP CPD 22](#_Toc318898234)

[Kód 5.10 Ukázkový výstup z aplikace phploc 23](#_Toc318898235)

[Kód 5.11: Ukázka možnosti využití PHPDoc (soubor phpdoc01.php) 24](#_Toc318898236)

# Rejstřík

# Přílohy

1. QA = quality assurance – kontrola kvality [↑](#footnote-ref-1)
2. Volně podle [Wikipedia contributors, 2011] [↑](#footnote-ref-2)
3. Přeloženo z [Fowler, 2006] [↑](#footnote-ref-3)
4. Viz http://www.extremeprogramming.org/rules/integrateoften.html [↑](#footnote-ref-4)
5. Zkracováno jako EP - viz [Beck, 2005] [↑](#footnote-ref-5)
6. SCM - Source Code Management Tools [↑](#footnote-ref-6)
7. Viz http://subversion.tigris.org/ [↑](#footnote-ref-7)
8. Concurrent Versions System – viz http://savannah.nongnu.org/projects/cvs [↑](#footnote-ref-8)
9. Viz https://jazz.net/projects/rational-team-concert/ [↑](#footnote-ref-9)
10. Viz http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server/overview [↑](#footnote-ref-10)
11. Viz http://git-scm.com/ [↑](#footnote-ref-11)
12. Viz http://mercurial.selenic.com/ [↑](#footnote-ref-12)
13. Viz http://bazaar.canonical.com/ [↑](#footnote-ref-13)
14. Viz http://www.bitkeeper.com/ [↑](#footnote-ref-14)
15. Viz http://kerneltrap.org/node/4966 [↑](#footnote-ref-15)
16. Získání kopie všech souborů z repositáře [↑](#footnote-ref-16)
17. Tzv. build [↑](#footnote-ref-17)
18. Java Development Kit http://www.oracle.com/technetwork/java/javase/downloads/index.html [↑](#footnote-ref-18)
19. Integrated development environment neboli Vývojové prostředí [↑](#footnote-ref-19)
20. Viz http://www.gnu.org/software/make/ [↑](#footnote-ref-20)
21. Viz http://ant.apache.org/ [↑](#footnote-ref-21)
22. Viz http://www.phing.info [↑](#footnote-ref-22)
23. TDD neboli vývoj řízený testy [Beck, 2002] [↑](#footnote-ref-23)
24. Viz http://www.martinfowler.com/bliki/Xunit.html [↑](#footnote-ref-24)
25. Viz http://www.junit.org/ [↑](#footnote-ref-25)
26. Viz http://www.nunit.org/ [↑](#footnote-ref-26)
27. Viz http://www.phpunit.de/ [↑](#footnote-ref-27)
28. Je důležité, aby integraci neprováděl na svém počítači, protože například pokud zapomene uložit nějaký soubor do úložiště, sestavení by prošlo (soubor má k dispozici), ale u ostatních členů týmu by selhalo (neměli by k dispozici chybějící soubor). [↑](#footnote-ref-28)
29. Viz kapitola 3.3 [↑](#footnote-ref-29)
30. Viz http://www.pragmaticautomation.com/cgi-bin/pragauto.cgi/Monitor/Devices/BubbleBubbleBuildsInTrouble.rdoc [↑](#footnote-ref-30)
31. http://dattein.com/blog/arduino-build-light/ [↑](#footnote-ref-31)
32. http://fabiopereira.me/blog/2009/12/15/build-dashboard-radiator-your-build-light-2/ [↑](#footnote-ref-32)
33. Viz <http://cruisecontrol.sourceforge.net/> [↑](#footnote-ref-33)
34. Viz <http://phpundercontrol.org/> [↑](#footnote-ref-34)
35. Viz <http://code.google.com/p/xinc/> [↑](#footnote-ref-35)
36. Viz <http://www.atlassian.com/software/bamboo/overview> [↑](#footnote-ref-36)
37. Viz <http://www.atlassian.com/> [↑](#footnote-ref-37)
38. Viz <http://www.urbancode.com/html/products/build/default.html> [↑](#footnote-ref-38)
39. Viz <http://www.jetbrains.com/teamcity/> [↑](#footnote-ref-39)
40. Viz <http://www.jetbrains.com/> [↑](#footnote-ref-40)
41. Viz <http://msdn.microsoft.com/en-us/vstudio/ff637362> [↑](#footnote-ref-41)
42. Viz <http://sismo.sensiolabs.org/> [↑](#footnote-ref-42)
43. Viz <http://travis-ci.org/> [↑](#footnote-ref-43)
44. Viz <http://www.sonarsource.org/> [↑](#footnote-ref-44)
45. Viz <http://docs.codehaus.org/display/SONAR/Hudson+and+Jenkins+Plugin> [↑](#footnote-ref-45)
46. Viz <http://news.cnet.com/8301-30685_3-20000019-264.html> [↑](#footnote-ref-46)
47. Viz <http://jenkins-ci.org/content/whos-driving-thing> [↑](#footnote-ref-47)
48. Viz <http://jenkins-ci.org/content/hudsons-future> [↑](#footnote-ref-48)
49. Viz <http://jenkins-ci.org/content/jenkins>, <http://jenkins-ci.org/> [↑](#footnote-ref-49)
50. Viz <http://kitscm.vse.cz/> [↑](#footnote-ref-50)
51. Viz http://www.zend.com/en/products/studio/ [↑](#footnote-ref-51)
52. Viz http://eclipse.org/pdt/ [↑](#footnote-ref-52)
53. Viz http://netbeans.org/features/php/ [↑](#footnote-ref-53)
54. Viz [Bergmann, 2008] [↑](#footnote-ref-54)
55. Viz http://blueparabola.com/blog/subversion-commit-hooks-php [↑](#footnote-ref-55)
56. Viz http://phpadvent.org/2008/dont-commit-that-error-by-travis-swicegood [↑](#footnote-ref-56)
57. Viz http://framework.zend.com/ [↑](#footnote-ref-57)
58. Vit http://pear.php.net/package/PHP\_CodeSniffer/ [↑](#footnote-ref-58)
59. Viz https://github.com/sebastianbergmann/phpcpd [↑](#footnote-ref-59)
60. Viz http://phpmd.org/ [↑](#footnote-ref-60)
61. Viz http://pmd.sourceforge.net/ [↑](#footnote-ref-61)
62. Viz https://github.com/sebastianbergmann/phploc [↑](#footnote-ref-62)
63. Viz <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html> [↑](#footnote-ref-63)
64. Viz <http://www.phpdoc.org/> nebo <http://pear.php.net/package/PhpDocumentor> [↑](#footnote-ref-64)
65. Viz <http://pear.php.net/bugs/bug.php?id=12128> [↑](#footnote-ref-65)
66. Viz <http://www.beranek.de/node/73> [↑](#footnote-ref-66)
67. Viz <http://www.docblox-project.org/> [↑](#footnote-ref-67)
68. Viz <http://zend-framework-community.634137.n4.nabble.com/ZF-API-documentation-td3451968.html#a3452171> [↑](#footnote-ref-68)
69. Viz <http://docs.docblox-project.org/for-users/validating-documentation-in-your-code.html#as-checkstyle-log-file> [↑](#footnote-ref-69)
70. Viz <http://apigen.org/> [↑](#footnote-ref-70)
71. Viz http://akrabat.com/zend-framework-tutorial/ [↑](#footnote-ref-71)
72. Viz http://framework.zend.com/ [↑](#footnote-ref-72)
73. Viz https://github.com/mhujer/zf-tutorial [↑](#footnote-ref-73)
74. Viz http://www.phing.info/docs/guide/stable/chapters/appendixes/AppendixC-OptionalTasks.html#PhpLintTask [↑](#footnote-ref-74)
75. Viz https://wiki.jenkins-ci.org/display/JENKINS/LTS+Release+Line [↑](#footnote-ref-75)
76. package repository [↑](#footnote-ref-76)
77. Viz https://wiki.jenkins-ci.org/display/JENKINS/Locale+Plugin [↑](#footnote-ref-77)
78. http://localhost:8080/configure [↑](#footnote-ref-78)
79. Viz https://wiki.jenkins-ci.org/display/JENKINS/Green+Balls [↑](#footnote-ref-79)
80. Viz https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin [↑](#footnote-ref-80)
81. Viz https://wiki.jenkins-ci.org/display/JENKINS/Phing+Plugin [↑](#footnote-ref-81)
82. Viz https://wiki.jenkins-ci.org/display/JENKINS/HTML+Publisher+Plugin [↑](#footnote-ref-82)
83. Viz https://wiki.jenkins-ci.org/display/JENKINS/Plot+Plugin [↑](#footnote-ref-83)
84. Viz https://wiki.jenkins-ci.org/display/JENKINS/JDepend+Plugin [↑](#footnote-ref-84)
85. Viz http://phpmd.org/documentation/creating-a-ruleset.html [↑](#footnote-ref-85)