Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Využití nástrojů pro správu verzí při vývoji software

David Holada

© 2017 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

David Holada

Informatika

Název práce

Využití nástrojů pro správu verzí při vývoji software

Název anglicky

Using version management systems in software development

Cíle práce

Cílem práce je popsat možnosti využití nástrojů pro správu verzí při vývoji software a prakticky na ukázkovém projektu demonstrovat možnost nasazení vybraného nástroje pro správu verzí a automatický deployment.

Metodika

Metodika práce je založena na analyticko-syntetickém přístupu. Bude provedeno studium a analýza odborných informačních zdrojů. Na základě syntézy zjištěných poznatků a praktických zkušeností budou shrnuty možnosti různých systémů pro správu verzí a vlastnosti vybraného systému budou demonstrovány prostřednictvím ukázkové aplikace systému na konkrétní softwarový projekt.

Doporučený rozsah práce

35-40 stran

Klíčová slova

GIT, SVN, CVS, verzovací systém, tým, nasazení, zdrojový kód

Doporučené zdroje informací

BRYAN O'SULLIVAN, Mercurial: The Definitive Guide, O'Reilly Media, 2009. ISBN: 978-0-596-80067-3.
MIKE MASON, Pragmatic Guide to Subversion, The Pragmatic Programmers, 2010. ISBN: 978-1-93435-661-6.
RAVISHANKAR SOMASUNDARAM, Git: Version Control for Everyone, Packt Publishing, 2013. ISBN: 978-1-84951-752-2.
SCOTT CHACON, Pro Git, Praha: CZ.NIC, 2009. ISBN: 978-80-904248-1-4.
TRAVIS SWICEGOOD, Pragmatic Guide to Git, The Pragmatic Programmers, 2010. ISBN: 978-1-93435-672-2.

Předběžný termín obhajoby

2016/17 ZS – PEF (únor 2017)

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 29. 01. 2017

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci „Využití nástrojů pro správu verzí při vývoji software“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 5. března 2017 ____________________________

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce Ing. Jiřímu Brožkovi Ph.D. za jeho odborné vedení a pomoc při zpracování této práce.

Využití nástrojů pro správu verzí při vývoji software

Using version management systems in software development

Souhrn: Tato práce pojednává o nástrojích pro správu verzí při vývoji software. Je zde představeno historické pozadí těchto systémů spolu s důvody jejich vzniku. Dále jsou uvedeny dnes nejpoužívanější z těchto nástrojů a provedena jejich základní komparace. Následně jsou předvedeny základní příkazy pro práci s verzovacími systémy. Pro demonstraci těchto příkazů je vybrán verzovací systém Git. V praktické části práce je hovořeno o aplikaci Gitu ve vývojovém prostředí Brackets za pomoci Git pluginu Brackets Git. Nakonec jsou porovnány možnosti pro práci s Gitem na serveru spolu s možnostmi automatického nasazování změn.

Summary: This thesis talks about version control sytems usage in terms of software development. There is introduced historical background of these systems together with reasons for their creation. Next are listed today's most used version control systems which are compared between each other. Further, there are showcased basic commands for work with version control systems. For demonstration of these commands is chosen version control system called Git. In practical part of this thesis there is talked about aplication of Git in Brackets IDE plugin Brackets Git. Finally, there are compared options for work with Git on server and options of auto-deployment.

Klíčová slova: Git, SVN, CVS, verzovací systém, tým, nasazení, zdrojový kód

Keywords: Git, SVN, CVS, version control system, team, deployment, source code

OBSAH

1 ÚVOD 10
2 CÍL PRÁCE A METODIKA 11
2.1 Stanovení cíle práce 11
2.2 Metodika 11
3 TEORETICKÁ ČÁST 13
3.1 DŮVOD VZNIKU VERZOVACÍCH SYSTÉMŮ 13
3.1.1 Efektivní ukládání více verzí software 13
3.1.2 Současná práce několika vývojářů na jednom kódu 14
3.1.3 Kontrola práce druhých vývojářů 14
3.2 HISTORIE VERZOVACÍCH SYSTÉMŮ 15
3.3 HISTORIE GITU 15
3.4 POROVNÁNÍ GITU A SVN 16
3.4.1 Decentralizace 16
3.4.2 Rychlost 17
3.4.3 GitHub a vývoj open source projektů 17
3.4.4 Velikost 18
3.4.5 Značení commitů 18
3.5 PRÁCE S GITEM 18
3.5.1 Instalace Gitu 18
3.5.2 Konfigurace Gitu 19
3.5.3 Vytvoření vlastního repozitáře 19
3.5.4 Naklonování cizího repozitáře 20
3.5.5 Nahrávání vlastních změn 20
3.5.6 Příkazy ve working directory 21
3.5.7 Příkazy ve staging area 22
3.5.8 Commit 23
3.5.9 Repository 24
3.5.10 Založení vzdáleného repozitáře pomocí služby Bitbucket 26
3.5.11 Přesun změn na BitBucket 29
3.5.12 SSH klíče 30

3.5.13 Týmová práce s gitem (větve) ... 33
3.5.14 Příkazy pro práci s větvemi ... 34
4 PRAKTICKÁ ČÁST ... 40
4.1 PRÁCE S GITEM VE VÝVOJOVÉM PROSTŘEDÍ ... 40
4.1.1 Vývojové prostředí Brackets ... 40
4.1.2 Plugin Brackets Git ... 41
4.1.3 Práce v Brackets Git ... 42
4.2 SOUČASNÉ MOŽNOSTI PRO PRÁCI S GITEM NA STRANĚ SERVERU ... 45
4.2.1 Klasické české webhostingy ... 45
4.2.2 Ekosystém obyčejného webhostingu, repozitářového uložiště a deployovací služby ... 48
4.2.3 Vlastní server ... 50
4.2.4 Virtuální/dedikované servery ... 51
4.2.5 Ekosystém repozitářového uložiště a cloudové platformy ... 53
5 ZHODNOCENÍ VÝSLEDKŮ A DOPORUČENÍ ... 55
6 ZÁVĚR ... 57
7 SEZNAM POUŽITÝCH ZDROJŮ ... 58

SEZNAM OBRÁZKŮ

Obr. 1 Schéma pohybu změn od uživatele k remote repository 20
Obr. 2 Vytváření účtu na Bitbucketu 26
Obr. 3 Bitbucket – přihlašovací formulář 27
Obr. 4 Prázdný výpis repozitářů s odkazem na vytvoření nového 27
Obr. 5 Tvorba nového repozitáře...28
Obr. 6 Výpis existujících repozitářů 29
Obr. 7 Bitbucket návod na propojení lokálního a vzdáleného repozitáře...............................30
Obr. 8 Přidání SSH klíče...31
Obr. 9 Přidání SSH klíče, 2. krok 32
Obr. 10 Lineární historie projektu 33
Obr. 11 Paralelní historie projektu...34
Obr. 12 Merge hell..39
Obr. 13 Instalace pluginu Brackets Git..41
Obr. 14 Brackets s otevřeným pluginem Brackets Git 42
Obr. 15 Brackets Git v režimu existujícího repozitáře 43
Obr. 16 Modální okno pro vytvoření commitu..43
Obr. 17 Brackets Git v režimu existujícího repozitáře 2 44
Obr. 18 Ceník plánů deployovací služby FTPLOY...49
Obr. 19 Ceník plánů deployovací služby DeployHQ 49

SEZNAM TABULEK

Tab. 1 Ceny dalších hostingů s podporou Gitu..47
Tab. 2 Ceny virtuálních a dedikovaných serverů u největších českých poskytovatelů.......52
Tab. 3 Porovnání cen GitHub vs BitBucket 53

SEZNAM GRAFŮ

Graf 1 Vývoj vyhledávání verzovacích systémů na Google dne Google Trends 16

1 ÚVOD

Ve vývoji softwaru, ostatně jako v každém jiném oboru, dochází postupem času k rozvoji nástrojů a prostředků, které ho činí stále rychlejším, efektivnějším a bezpečnějším. Jedněmi z těchto nástrojů jsou čím dál tím více skloňované verzovací systémy, které si za posledních několik let našly své nezastupitelné místo ve vývojářských kruzích a jejichž obliba stále roste.

Tyto nástroje vzešly z potřeby vývojářů pohybovat se v historii napsaného kódu a možnosti pracovat efektivně v týmu, pokud vývojáři pracovali na stejném projektu. Oba tyto problémy jsou bez verzovacích systémů jen těžko řešitelné. Verzovací systémy tyto problémy velice elegantním způsobem vyřešily.

V dobách neexistence verzovacích systémů byl totiž jediný způsob pohybu mezi verzemi používat zálohy a práce několika lidí současně na jednom projektu, byl prakticky neřešitelný problém. Tyto problémy tak už dnes naštěstí nemusí žádný vývojář používající verzovací systémy řešit.

Mezi nejznámější verzovací systémy patří v současné době Git a Subversion SVN. V této práci budou pro porovnání zmíněny všechny, avšak hlavní pozornost bude věnována Gitu jakožto nejpopulárnějšímu verzovacímu systému dnešní doby.

2 CÍL PRÁCE A METODIKA

2.1 STANOVENÍ CÍLE PRÁCE

Cílem této práce je podat informace o tom, co to jsou verzovací systémy, jak fungují a k čemu slouží. Následně budou vybrány nejznámější zástupci z těchto systémů, které mezi sebou budou porovnány z různých hledisek.

V teoretické části bude vybrán jeden ze systémů, který bude podrobněji popsán a na kterém budou předvedeny nejběžněji používané příkazy pro práci se soubory. Pro účely této práce bude tímto systémem v současnosti nejvíce populární Git.

V praktické části bude předvedeno používání Gitu při běžné práci na vývoji softwaru, kde bude Git spouštěn z grafického rozhraní pluginu Brackets Git textového editoru Brackets, který implementuje nejvíce používané Gitovské příkazy do grafického rozhraní. Kromě implementace Gitu lokálně, tedy v pluginu textového editoru, budou rozebrány možnosti práce s Gitem na straně serveru. Jednotlivá řešení mezi sebou budou vzájemně porovnány jak z hlediska nabízených možností, tak ceny. Nakonec budou doporučena nejlepší řešení pro konkrétní uživatele.

2.2 METODIKA

Bakalářská práce bude zpracována na základě uvedených literárních a internetových zdrojů v syntéze s vlastními znalostmi získanými praxí ve vývojářských firmách. Dále budou uplatněny postřehy z odborných přednášek a školení s Gitovskou tematikou. Díky těmto pramenům a znalostem bude možné podat souhrnné informace o dané problematice. Pro spouštění demonstračních příkazů bude použit operační systém Ubuntu 14.04 Thrusty Tahr jakožto distribuce Linuxu, čili Gitovské nativní prostředí. Použitá statistická tvrzení budou doložena dle grafů či tabulek Google Trends a CZ.NIC. Použité obrázky budou buď vlastními výstřižky z používaných služeb či vývojového prostředí, v opačném případě budou řádné ozdrojovány.

V úvodu praktické části bude použito vývojové prostředí Brackets ve verzi 1.8 taktéž na operačním systému Ubuntu 14.04 Thrusty Tahr. Při porovnávání jednotlivých služeb pro práci s Gitem na serveru budou použity aktuální informace a aktuální ceníky ze stránek poskytovatelů služeb ke dni zpracování. Tyto ceny budou uvedeny včetně 21%

DPH. Pokud budou ceny uvedeny v jiné měně, kurz těchto měn bude přejat ke stejnému dni.

3 TEORETICKÁ ČÁST

3.1 DŮVOD VZNIKU VERZOVACÍCH SYSTÉMŮ

Verzovací systémy vznikly z naprosto přirozené potřeby vývojářů uchovávat v rozumné formě více verzí softwaru čili obrazů kódu v daném časovém okamžiku, a hlavně potřeby pracovat v týmu spolu s dalšími vývojáři na jednom zdrojovém kódu jednoho projektu současně.

3.1.1 Efektivní ukládání více verzí software

První problém čili uchovávání více verzí softwaru, byl tím menším. Tento problém se totiž dal, byť ne moc efektivně, řešit. V dobách, kdy ještě neexistovaly verzovací systémy, vývojáři používali k uchování starších verzí kódu systém záloh. Tento systém je znám asi každému běžnému uživateli počítače. Pro úplnost zde však bude raději popsán.

Jako příklad pro vysvětlení může posloužit vývojář píšící program efektivně implementující šifrování souborů nějakou šifrou. Vývojář začne psát první verzi programu, ale zjistí, že program pracuje příliš pomalu, zkusí tedy alternativní postup, který by mohl být potencionálně rychlejší. Aby však nepřišel o původní verzi, která by se nakonec přesto mohla ukázat jako rychlejší než verze nová, vytvoří si někde na disku složku „zaloha“ a do ní zkopíruje aktuální zdrojové kódy programu. Tímto způsobem si uchovává každou verzi, která obsahuje nějaký potencionálně užitečný kód, aby o něj nepřišel. Problém však je, že čím více variant zkouší, tím více místa pro zálohy potřebuje. Tento fakt je navíc umocněn tím, že každá nová verze je pravděpodobně více datově náročná než ta předchozí. Pokud je tímto způsobem vyvíjen software ve velkém počtu iterací, začíná rychle ubývat místo na disku.

Dnes by pravděpodobně běžnému uživateli tento problém s místem na disku při velikosti dnešních běžně vyráběných disků asi nevadil, ale ještě před pár desítkami let, kdy byla kapacita vyráběných uložišť silně nedostačující, mohla být nepříjemným limitem. Samozřejmě tu ale nejsou pouze běžní uživatelé, pokud bude problém převeden např. na dnešní cloudová uložiště kódu, která již efektivní ukládání kódu při počtu uživatelů, jímž prostor poskytují, řešit opravdu musí, tak existence nějakého nástroje typu verzovacích

systémů, který dokáže uložit stejná data na méně místa, jim pak pochopitelně šetří náklady na pořízení pevných disků a tím zvyšuje zisk, což je cílem každého podniku. Verzovací systémy tak nemají přínos pouze vývojářský, ale také finanční.

3.1.2 Současná práce několika vývojářů na jednom kódu

Druhý problém, a sice současná práce několika vývojářů na jednom kódu, je už bez verzovacích systémů prakticky neřešitelný.

Může totiž nastat situace, kdy potřebují dva vývojáři pracovat společně na jednom projektu a oba, byť každý vytváří jinou „featuru“, zasahují do stejných souborů. Zde nastává velký problém, jakmile oba začnou upravovat stejný soubor ve stejnou chvíli. V tom okamžiku musí totiž čelit tzv. „souběžnému přístupu“.

Jeden vývojář otevře soubor a začne dělat svoje změny, mezitím stejný soubor otevře druhý vývojář a začne dělat také svoje změny. Jakmile jeden z nich dokončí svoji práci, soubor uloží. Poté skončí práci i druhý vývojář a soubor také uloží, tím však nahraje na server svojí verzi a přepíše všechny změny prvního vývojáře.

Toto vývojáři mohli vyřešit jen dobrou vzájemnou komunikací. Např. pokud věděli, že budou ve stejnou chvíli upravovat stejné soubory, mohli se např. domluvit, že jeden bude pracovat přes den a druhý přes noc. Tím se však rapidně brzdil vývoj, protože jeden vývojář musel čekat, než ten druhý skončí se svou prací. Tento systém má sice velké nedostatky, ale v hodně malém týmu je ještě jakž takž použitelný. Ve velkém týmu je však nepoužitelný a zde neexistuje jiná možnost než použití verzovacích systémů.

3.1.3 Kontrola práce druhých vývojářů

Kontrolu práce druhých vývojářů snad ani nelze počítat jako přímo důvod pro vznik verzovacích systémů, nicméně je to příjemný bonus, který vyplynul z ukládání historie verzí.

Při vývoji software se totiž skupina vývojářů v praxi lehce dostane do bodu, kdy některý z nich udělá chybu a svou úpravou znefunkční nebo v tom horším případě dokonce byť třeba nedopatřením smaže již něco funkčního, co naprogramoval jiný vývojář. Tato chyba však může být skryta několik týdnů či dokonce měsíců, protože při testování nového kódu se málokdy testuje celá aplikace, ale většinou jen základní a nově přidaná

funkcionalita. Správně by se sice měla po každé úpravě testovat znovu celá aplikace, to je však z hlediska ekonomických nákladů v praxi nemyslytelné. Vzniklá chyba pak jen čeká až někdo po X dnech přijde a použije kombinaci úkonů, při které se chyba projeví. V tu dobu je reakce všech vývojářů většinou ve stylu: „Vždyť to doteď fungovalo a nikdo s tím nic nedělal". Samozřejmě vždy s tím „někdo něco dělal", jinak by chyba nemohla vzniknout. Problém se tak začne řešit, že šéf jako první vyhledá vývojáře, který danou funkcionalitu programoval, byť za chybu nemůže a chudák vývojář už jen těžko může vysvětlovat, že chybu nezpůsobil.

S verzovacími systémy je však tento trochu směšný příklad, který však v mnohých firmách funguje doteď, nepředstavitelný. Při používání verzovacích systémů lze snadno vyvolat historii uživatelů, kteří pracovali v minulosti na kódu souvisejícím se vzniklou chybou a během pár minut je všem jasné, kdo a kdy udělal chybu. A tak nedochází ke zbytečným dohadům, domněnkám, zapírání a obviňování, které chodu firmy rozhodně neprospívají.

3.2 HISTORIE VERZOVACÍCH SYSTÉMŮ

1972 - Source Code Control System (SCCS) první verzovací systém, komerční model *(Prskavec, 2011)*

1981 - Revision Control System (RCS) open source náhrada za SCCS *(Prskavec, 2011)*

1990 - Concurrent Versions System (CVS) první dnes již známější verzovací systém, nicméně pro dnešní dobu stále nepoužitelný, trpěl spoustou nevýhod např. nemožnost přejmenování složek (neverzování složek) *(Trenz, Kolomazník, 2011)*

2000 - BitKeeper, Subversion už lze verzovat celý strom adresářů, ne pouze soubory *(MASON, 2010)*

2005 - Git a Mercurial *(O'SULLIVAN, 2009)*

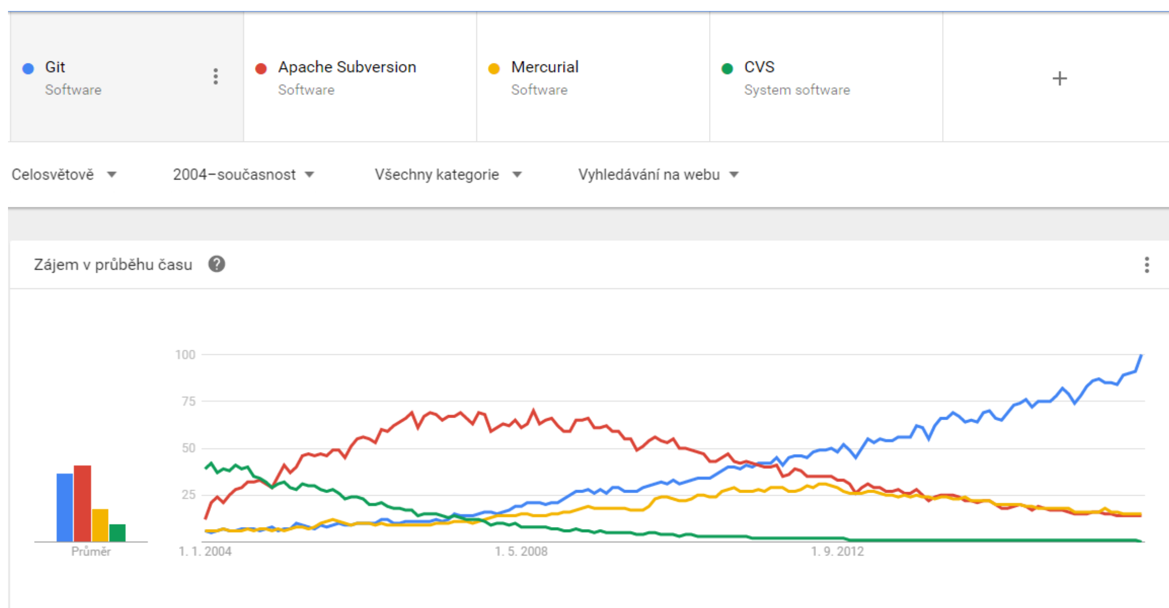
3.3 HISTORIE GITU

Historie Gitu se začíná psát v roce 2005, kdy přešel tehdejší verzovací systém BitKeeper na placený model, což se ale nelíbilo komunitě kolem Linuxu vedené Linusem Torvaldsem, která do té doby používala BitKeeper k verzování jádra Linuxu.

Komunita kolem Linuxu a hlavně sám Linus Torvalds se tedy rozhodli napsat vlastní a lepší verzovací systém založený na poznatcích zjištěných při používání BitKeeperu a ještě ten samý rok vyšla první verze Gitu *(MCMILLAN, 2005)*.

Git se stal postupem času své existence nejpopulárnějším verzovacím systémem na světě a dnes již funguje pod verzí 2. 12. 0 *(Git Official Website, 2017)*.

Dle Goole Trends *(Graf 1)* je vidět postupná jasná dominance Gitu nad konkurencí.



Graf 1 Vývoj vyhledávání verzovacích systémů na Google dne Google Trends
Zdroj: vlastní zpracování

3.4 POROVNÁNÍ GITU A SVN

3.4.1 Decentralizace

Hlavní výhodou Gitu oproti SVN je zcela jistě decentralizace, to znamená, že každý, který na projektu pracuje, má u sebe kopii celého repozitáře *(CHACON, 2009)*.

Z toho plynou hlavní 3 výhody:

1) Pokud probíhá práce na repozitáři bez přístupu k internetu, např. někde na cestách není s Gitem žádný problém. Existence lokální kopie repozitáře totiž dovoluje bez problému pracovat i bez přístupu k internetu a jakmile je

připojení k internetu k dispozici, změny se mohou sloučit do vzdáleného repozitáře *(GitSvnComparison, 2013).*

2) Pokud se z nějakého důvodu poškodí vzdálený repozitář, je možnost ho vždy obnovit z kopií všech, kteří na projektu pracovali. Pokud se toto stane u nedecentralizovaného verzovacího systému a neexistují jeho pravidelné zálohy, není možnost jak ho obnovit a všechna práci je tak v tu chvíli nenávratně pryč.
3) Pokud dojde k rozbití kódu ve vzdáleném repozitáři, vývojáři, pokud používají Git, nemusí čekat, než se chyba opraví, aby mohli dál pracovat, ale pokračují si mezi tím na lokální kopii (repozitáři) a jakmile je vyřešen problém ve vzdáleném repozitáři, mohou do něj opět bez problému zanést svoje lokální změny.

Samozřejmě decentralizace přináší i nevýhodu.

1) Jelikož se nepracuje s jedním centrálním repozitářem, ale s lokálním a vzdáleným repozitářem, může být pro začátečníky problém pochopit logiku toho, jaké akce se provádí s lokálním a jaké se vzdáleným repozitářem, o to víc pro uživatele, kteří jsou již z dřívější doby zvyklí na logiku jednoho repozitáře z nedecentralizovaných systémů.

3.4.2 Rychlost

Jelikož je u Gitu většina operací lokálních, není zde žádná latence mezi serverem, a tak je o mnoho rychlejší než nedecentralizované systémy jako právě SVN *(LOURDAS, 2011).*

3.4.3 GitHub a vývoj open source projektů

Vzhledem k decentralizaci Gitu mohla vzniknout úložiště repozitářů typu GitHub, která umožňují pohodlný vývoj open source projektů.

Každý si odtud může stáhnout (forknout) kopii jakéhokoliv zde uloženého open source projektu a na ní něco nového naprogramovat nebo přeprogramovat stávající. Jakmile má hotovo, požádá o tzv. pull request, kterým odešle svoje změny vlastníkovi

repozitáře, ve kterém dělal úpravy. Ten si je pak může prohlédnout a následně rozhodnout, zda je do hlavního repozitáře začlení či nikoliv.

Tento princip je podmíněný decentralizací, v čemž SVN nemá jak konkurovat, jelikož se v jeho případě jedná o nedecentralizovaný verzovací systém *(GitHub Help, 2017).*

3.4.4 Velikost

Git hlavně při rozsáhlých projektech podstatně šetří místo, např. repozitář Mozilly v Gitu je 40x menší než v SVN *(GitSvnComparison, 2013).*

3.4.5 Značení commitů

SVN značí jednotlivé commity čísly popořadě od 1 do X, Git ke značení commitů naopak používá 40 místné hexadecimální hashe *(How is git commit sha1 formed, 2012).* Z tohoto plyne pro SVN jasná výhoda přehlednosti, nemusí se totiž pro přechod na jiný commit hledat hash označení chtěného commitu, ale stačí vědět, který commit je právě aktuální a z něj se lehko odvodí (přičte či odečte) číslo požadovaného commitu.

Nicméně Git by bez složitých hashů nemohl fungovat, neobsahují totiž pouze pořadí commitu, ale ukládají v sobě i informace o autorovi, rodičovském a zdrojovém commitu, ale také o samotné commit message, v čemž je Git opravdu geniální.

3.5 PRÁCE S GITEM

3.5.1 Instalace Gitu

Git jako každý jiný software je potřeba nejprve nainstalovat. Tento krok se liší podle toho, na jakém OS je instalace prováděna *(SOMASUNDARAM, 2013).*

Instalace na Windows:

Instalace na Windows probíhá velice podobně jako všechny ostatní instalace. Stačí stáhnout instalační .exe soubor z adresy https://git-for-windows.github.io/, proklikat se instalací a po dokončení je k dispozici jak Git v příkazové řádce, tak Git v grafickém uživatelském rozhraní *(Git for Windows, 2017).*

Instalace na Linuxu:

Instalace na Linuxu probíhá ještě mnohem rychleji než na Windows, stačí spustit příkaz balíčkovacího systému:

```
apt-get install git
```

případně

```
yum install git
```

dle toho, na jaké distribuci linuxu instalace zrovna probíhá a tím je hotovo *(CGACON, 2009).*

3.5.2 Konfigurace Gitu

Po instalaci je nutné, aby byl Git správně nakonfigurován. Jedná se vlastně jen o nastavení pár proměnných.

Pro identifikaci uživatele je nutné nastavit proměnné:

```
user.name
user.email
```

bez jejich nastavení Git stejně nepovolí jakékoliv práce v repozitáři, protože práce v něm musí být spojena s jasně identifikovaným uživatelem. Toto natavení může vypadat např. Takto:

```
git config --global user.name "Pavel Novák"
git config --global user.email pavel.novak@domena.cz
```

Pozn.: přepínač „--global" slouží k nastavení proměnných pro všechny repozitáře, jeho vynecháním by se nastavili odlišné údaje pro aktuální repozitář *(Git documentation, 2017).*

3.5.3 Vytvoření vlastního repozitáře

Aby Git začal verzovat soubory např. nějakého projektu, je nutné v jeho kořenovém adresáři spustit příkaz:

```
git init
```

Tento příkaz incializuje repozitář, a dá tak možnost použít příkaz *(Git Documentation, 2017)*:

```
git add
```

který umožňuje opravdu doslova přidat soubory, jejichž změny chceme sledovat. V praxi se často používá příkaz:

```
git add .
```

který začne sledovat změny ve všech souborech aktuální složky.

3.5.4 Naklonování cizího repozitáře

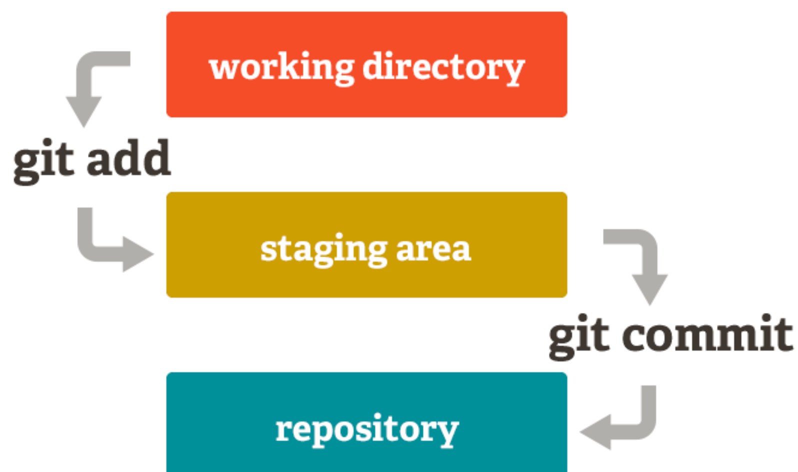
Pokud je potřeba začít pracovat s již existujícím repozitářem, nabízí Git příkaz *(SOMASUNDARAM, 2013)*:

```
git clone https://github.com/uzivatel/repositar.git
```

Pozn.: Tento příkaz naklonuje repozitář z jakékoliv URL, v ukázce je použita smyšlená adresa z úložiště repozitářů GitHub.

3.5.5 Nahrávání vlastních změn

Bezprostředním krokem při používání Gitu, jakmile je zprovozněn repozitář, je situace, kdy uživatel začne pracovat se soubory, tzn., že dojde k jejich editaci, vytváření nových či mazání těch již nežádoucích. Zde přichází na řadu Gitovská filozofie práce se změnami v repozitáři *(Obr. 1)*.



Obr. 1 Schéma pohybu změn od uživatele k remote repository
Zdroj: http://newtfire.org/dh/git_shell/gitWorkflow.jpg

Na rozdíl od SVN Git neukládá změny rovnou do jednoho společného centrálního repozitáře, ale používá o něco komplikovanější způsob, který vyplývá přímo z logiky decentralizovaných verzovacích systémů.

Pro pochopení problému je potřeba vysvětlit 4 základní pojmy, se kterými Git pracuje:

1) **Working directory**

Změny souborů v projektu v lokálním prostředí (na PC) oproti poslednímu commitu.

2) **Staging area**

Změny z working directory připravené pro další commit.

3) **Commit**

Souhrn uživatelových změn v souborech v logickém celku.

4) **Repository (remote repository)**

Stav souborů ve vzdáleném (remote) repozitáři na serveru. Obsahuje nahrané změny v souborech od všech uživatelů projektu.

3.5.6 Příkazy ve working directory

Změny ve working directory vznikají automaticky, jakmile jsou provedeny změny ve sledovaných souborech. Sledované soubory jsou všechny soubory přidané pomocí příkazu „git add" již zmíněného v sekci „Vytvoření vlastního repozitáře". Do working directory se taktéž automaticky přidávají všechny nově vytvořené nebo smazané soubory *(CHACON, 2009)*.

Stav working directory lze vždy vyvolat příkazem:

```
git status
```

Pokud by tedy nastala situace, kdy uživatel udělal změnu v již sledovaném souboru např. file.txt, výstup příkazu „git status" by vypadal takto:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
```

```
        modified:   file.txt
```

Z tohoto výpisu lze vyčíst soubory ve working directory jako „Changes not staged for commit", čili změny, které ještě neprošli do „staging area".

3.5.7 Příkazy ve staging area

Dalším krokem pro přesun vytvořených změn do repozitáře, je tyto změny v souborech uchovávané právě ve working directory připravit do nějakého logického celku se změnami, který se nazývá commit a právě k tomuto účelu slouží staging area *(SWICEGOOD, 2010)*.

Do staging area se přesouvají všechny takovéto soubory příkazem:

```
git add
```

Pokud chce tedy uživatel přesunout změny v souboru file.txt do staging area, použije příkaz:

```
git add file.txt
```

Nyní může opět pomocí příkazu:

```
git status
```

ověřit, zda došlo k přesunutí tohoto souboru do staging area:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file.txt
```

V tomto zápisu je již vidět, že se modifikovaný soubor file.txt přesunul ze sekce „Changes not staged for commit" do sekce „Changes to be commited". Nyní jsou tedy změny připraveny k přesunutí do vzdáleného repozitáře na serveru, k čemuž slouží commit.

3.5.8 Commit

Termín commit se používá ve 2 významech:

Commit jako logický celek změn:

Neboli souhrn všech souborových úprav uživatele, které souvisí s nějakým úkolem, na kterém pracuje. Commit je jasně identifikovatelný hexadecimálním hashem, autorem a časem vytvoření. Jednotlivé commity jsou tedy obrazy projektu v jeho určité fázi. Každý další commit většinou obohacuje commit předchozí *(CHACON, 2009)*.

V projektu by mohl výpis commitů vypsaný přes příkaz:

```
git log
```

vypadat např. nějak takto:

```
commit 5d424e3c58653a8dd05f3545156d506c4668d4d5
Author: Karel Novák <user@email.com>
Date:   Wed Nov 16 20:43:56 2016 +0200

    Gallery template

commit 905dbd2702e5d4d5fffd7f397b928e704dcc79fb
Author: Karel Novák <user@email.com>
Date:   Wed Nov 16 15:43:14 2016 +0200

    Subpage template

commit 91cab974d931782d12ed1bef2b24aaf185750053
Author: Karel Novák <user@email.com>
Date:   Wed Nov 16 13:24:59 2016 +0200

    Homepage template

commit f7f6815a29e53f073af4816ceaa62fca63d2cb95
Author: Karel Novák <user@email.com>
Date:   Wed Nov 16 10:08:25 2016 +0200
```

```
Initial commit
```

Z tohoto zápisu můžeme vyčíst, že uživatel pravděpodobně připravoval šablony pro webovou stránku. V prvním commitu uživatel inicializoval projekt, např. nahrál nějakou základní strukturu projektu, aby mohl v druhém commitu připravit šablonu úvodní stránky, ve třetím podstránky a v posledním galerie.

Příkaz „git log“ samozřejmě umí historii commitů vypisovat i mnohem přehledněji, defaultně však vypisuje tímto způsobem *(Git Documentation, 2017)*.

Commit jako příkaz:

Aby mohl být commit ve smyslu celku změn vytvořen, obsahuje git i stejnojmenný příkaz:

```
git commit
```

Zavoláním tohoto příkazu uživatel vytvoří commit se změnami ve staging area. Pokud by tedy chtěl uživatel vytvořit commit obsahující změny v souboru file.txt z předešlé ukázky, vypadal by výpis příkazu nějak takto:

```
$ git commit -m 'Changes in file.txt'
[master 83e38c7]  Changes in file.txt
 1 files changed, 8 insertions(+), 2 deletions(-)
```

Na příkladu je vidět za commitem použití parametru „-m“. Tento parametr slouží pro přidání popisu commitu, který by se měl týkat prováděných změn a který musí být součástí každého commitu.

Poslední řádek už jen informuje, že byl v commitu změněn 1 soubor a došlo u něj k přidán znaků na 8 řádcích a odebrání znaků na 2 řádcích.

3.5.9 Repository

V tomto stavu je sice již commit úspěšně vytvořen, nicméně ostatní uživatelé o změnách stále nevědí. Commit je totiž stále pouze v lokálním prostředí u uživatele, jež commit vytvořit. Pro nahrání změn na vzdálený repozitář slouží příkaz *(SWICEGOOD, 2010)*:

```
git push
```

Po jeho zavolání nastane podobný výpis jako zde:

```
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 255 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To ssh://git@bitbucket.org/user/webpage.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Nyní jsou změny úspěšně nahrány na serveru a může si je stáhnout do projektu jakýkoliv jiný uživatel, který má do repozitáře přístup.

Toto je však případ, pokud uživatel commitoval do repozitáře, který byl získán přes „git clone“ nebo-li naklonován a měl do něj již přístup. Pokud byl však uživatel zároveň tvůrcem repozitáře, čili ho na začátku inicializoval přes „git init“, git mu vrátí tuto hlášku:

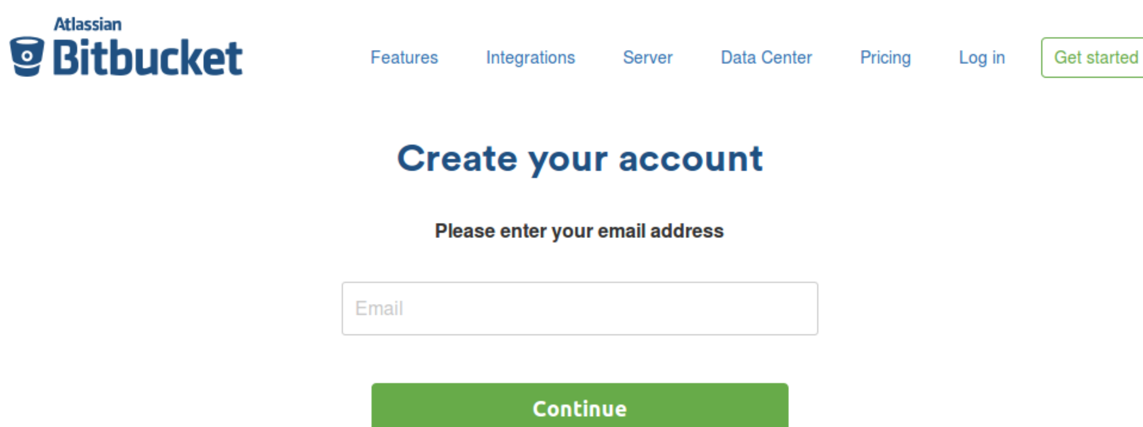
```
fatal: No configured push destination.
Either specify the URL from the command-line or configure a remote repository using

    git remote add <name> <url>

and then push using the remote name

    git push <name>
```

To znamená, že repozitář ještě nebyl propojen se vzdáleným repozitářem, do kterého se nahrávají změny, aby si je mohli stáhnout i ostatní uživatelé.

Vzdálený repozitář jde založit buď na vlastním serveru, který podporuje Git nebo lze využít služeb třetích stran, což je pro začínajícího uživatele rozhodně snadnější způsob. Mezi nejznámější z těchto služeb patří GitHub a Bitbucket. Jejich porovnání a hlubší analýza bude provedena v pozdější části práce. Zde bude předvedeno založení repozitáře ve službě Bitbucket.

3.5.10 Založení vzdáleného repozitáře pomocí služby Bitbucket

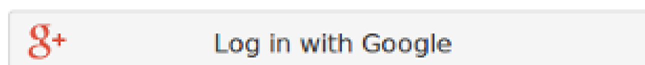
Služba BitBucket se nachází na adrese https://bitbucket.org. Jako první si zde uživatel musí založit svůj účet. To je možné na adrese https://bitbucket.org/account/signup/ *(Obr. 2)*.



Obr. 2 Vytváření účtu na Bitbucketu
Zdroj: vlastní zpracování

Zde uživatel vyplní svůj e-mail, následně zvolí uživatelské jméno a heslo a počká, až mu přijde na e-mail, pod kterým se registroval, e-mail s verifikačním odkazem. Po kliknutí na verifikační odkaz se uživatel již může přihlásit na adrese https://bitbucket.org/account/signin/ *(Obr. 3)*.

Po přihlášení je uživatel přesměrován na hlavní stránku, tzv. dashboard na adrese https://bitbucket.org/dashboard/overview. Zde jsou v pravém panelu vidět všechny uživatelovy repozitáře. V tomto případě však uživatel ještě žádný repozitář nevytvořil, proto BitBucket v pravém panelu místo výpisu existujících repozitářů zobrazí hlášku „Your repository list is empty“ a odkaz na vytvoření prvního repozitáře „Create a repository“ *(Obr. 4)*.



or

Email

Password

Forgot your password?

Need an account? Sign up.

Obr. 3 Bitbucket – přihlašovací formulář
Zdroj: vlastní zpracování

Your repository list is empty

Repositories that you own or watch will show up here.

Create a repository

Obr. 4 Prázdný výpis repozitářů s odkazem na vytvoření nového
Zdroj: vlastní zpracování

Po kliknutí na tento odkaz se otevře nová stránka s formulářem pro vytvoření nového repozitáře na adrese https://bitbucket.org/repo/create *(Obr. 5)*.

Zde je povinné vyplnit název repozitáře jako „Repository name“. Tento název by měl vystihovat projekt, na kterém uživatel pracuje.

Dále lze zvolit „Access level“ čili zda bude repozitář soukromý či nikoliv. Pokud tedy chce uživatel na repozitáři pracovat s více uživateli, kde konec konců Git vyniká nejvíce, nechá toto pole odškrtnuté. Pokud však chce verzovat nějaký soukromý projekt, pole zaškrtne.

Jako poslední lze vybrat, jakým verzovacím systémem bude repozitář využíván. BitBucket nabízí Git a Mercurial. Mercurial byl zmíněn již v úvodu práce, jelikož jsou však všechny ukázky prezentovány na Gitu, je nutno zaškrtnout právě možnost „Git“.

V Advanced settings pak může uživatel nastavit např. popis projektu či programovací jazyk, ve kterém bude projekt psán a spoustu dalších, zatím nepodstatných údajů.

Create a new repository
Import repository
Repository name*
Access level
This is a private repository
Repository type
Git
Mercurial
Advanced settings
Create repository
Cancel

Obr. 5 Tvorba nového repozitáře
Zdroj: vlastní zpracování

Po kliknutí na tlačítko „Create repository“ dojde k vytvoření vzdáleného repozitáře. Na úvodní stránce Bitbucketu na adrese https://bitbucket.org/uzivatelske-jmeno pak již může uživatel vidět místo hlášky „Your repository list is empty“ výpis existujících repozitářů, v tomto případě pouze jednoho repozitáře, vytvořeného v předchozím kroku. Tento repozitář bude obsahovat soubory webové stránky, byl proto pojmenován „webovaStranka“, záměrně camel casem bez diakritiky, jelikož Bitbucket diakritiku stejně ořezává a název „Webová stránka“ by transformoval na nehezké „webov-str-nka“. Grafické zobrazení výpisu demonstruje následující obrázek *(Obr. 6)*.

Obr. 6 Výpis existujících repozitářů
Zdroj: vlastní zpracování

3.5.11 Přesun změn na BitBucket

Po rozkliknutí tohoto repozitáře Bitbucket sám zobrazí krátkou nápovědu, co dál dělat s nově vytvořeným repozitářem. Aby mohl uživatel propojit lokální repozitář se vzdáleným, bude ho zajímat část „Get started with command line“ a podčást „I have an existing project“ *(Obr. 6)*.

Prvním krokem je samozřejmě přepnutí se do adresáře s projektem, kde již existuje lokální repozitář. V případě linuxového operačního systému tak bude cesta nejpravděpodobněji takto:

```
cd /var/www/webovaStranka
```

Nyní je vše připraveno k použití klíčového příkazu, který propojí lokální repozitář se vzdáleným umístěným na Bitbucketu:

```
git remote add origin
ssh://git@bitbucket.org/holdav/webovastranka.git
```

Tento příkaz nastaví lokálnímu repozitáři cestu ke vzdálenému repozitáři a pojmenuje ho jako „origin“. Může ho pojmenovat sice jak chce, ale toto je dobrá zažitá konvence.

Get started with command line

I have an existing project

Step 1: Switch to your repository's directory

```
1  cd /path/to/your/repo
```

Step 2: Connect your existing repository to Bitbucket

```
1  git remote add origin ssh://git@bitbucket.org/holdav/webovastranka.git
2  git push -u origin master
```

Obr. 7 Bitbucket návod na propojení lokálního a vzdáleného repozitáře
Zdroj: vlastní zpracování

Posledním krokem celého cyklu přesunu změn od uživatele na vzdálený repozitář na serveru je příkaz push.

Tento příkaz nahraje uživatelem vytvořené commity na vzdálený repozitář. Pokud uživatel používá příkaz push poprvé, měl by použít parametr „-u“, konec konců to samé radí Bitbucket nápověda *(Obr. 7)*:

```
git push -u origin master
```

Tento parametr nastaví výchozí chování pushe, a tak uživatel při dalších pushích již nemusí volat

```
git push origin master
```

ale čistý push:

```
git push
```

který se sám přeloží na:

```
git push origin master
```

Tento zápis přeloženě znamená, že se nahraje obsah hlavní větve „master“ do vzdáleného repozitáře pojmenovaného „origin“ *(Git Documentation, 2017)*.

3.5.12 SSH klíče

Při prvním pushi do vzdáleného repozitáře však dojde ještě k jedné zajímavé věci. Git totiž při prvním uživatelově pushi nedovolí přístup do repozitáře a zahlásí tuto hlášku:

```
Warning: Permanently added the RSA host key for IP address
```

```
'104.192.143.1' to the list of known hosts.
Permission denied (publickey).
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

První 2 řádky říkají, že se uživatel poprvé připojuje k serveru, kde je umístěný vzdálený repozitář. Git z bezpečnostních důvodů zobrazí hlášku, že se připojuje k serveru, ke kterému se ještě nikdy před tím nepřipojoval a automaticky ho zařadí do whitelistu.

Toto slouží uživateli jako kontrola, kam se připojuje. Pokud by např. věděl, že se k danému serveru již připojoval a zobrazila se mu tato hláška, ví, že je něco špatně a může začít včas řešit bezpečnostní riziko.

Zbylé řádky pak říkají, že nemohlo dojít k připojení na server, protože serveru není znám uživatelův veřejný klíč. Vzdálený repozitář totiž funguje přes protokol ssh, který umožňuje autentizaci pomocí veřejných a soukromých klíčů. Uživatel tedy musí nejprve na svém počítači tento pár vygenerovat a veřejný klíč poté uložit na server, ke kterému se připojuje. V tomto případě se jedná o zadání klíče na bitbucket.org.

K vytvoření páru SSH klíčů slouží příkaz *(GitHub Help: Authenticating to GitHub, 2017)*:

```
ssh-keygen -t rsa -b 4096 -C "pavel.novak@domena.cz"
```

Po jeho zavolání se vygeneruje veřejný klíč do umístění:

```
~/.ssh/id_rsa.pub
```

Odtud je potřeba klíč zkopírovat ať už za pomocí příkazové řádky nebo systémového GUI a vložit na server. Na Bitbucket.org k tomuto účelu slouží adresa https://bitbucket.org/account/user/user-name/ssh-keys/ a zde tlačítko „Add key" *(Obr. 8)*.

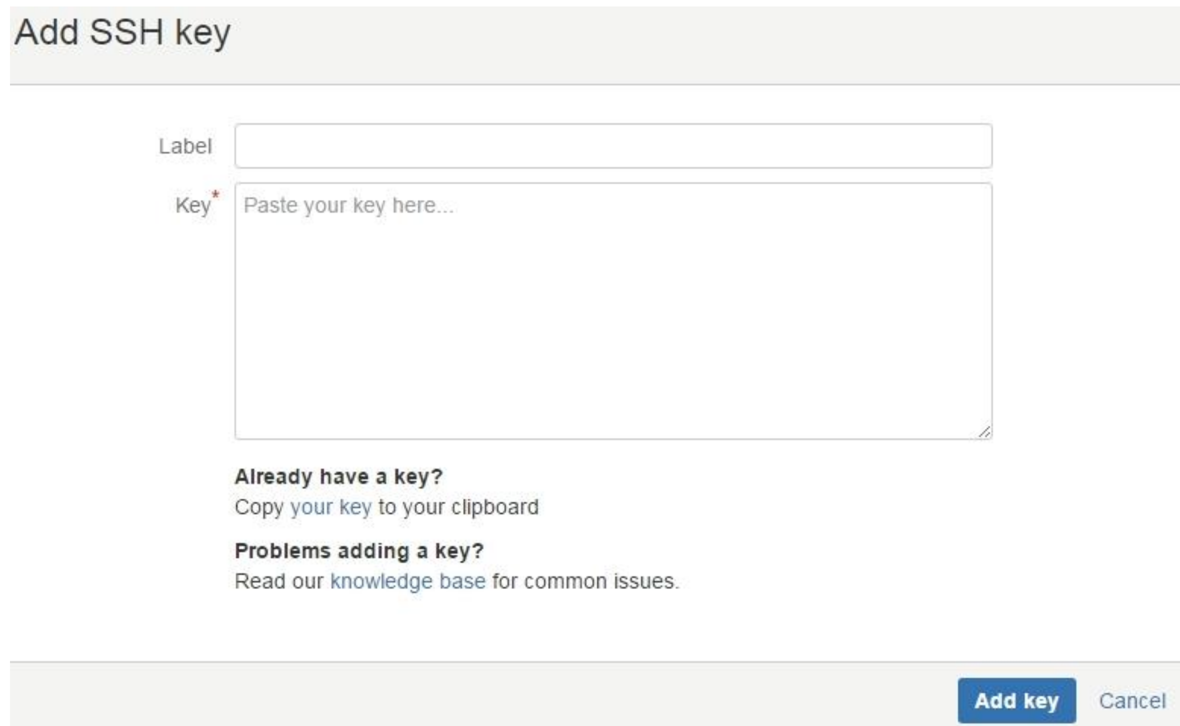
SSH keys

Use SSH to avoid password prompts when you push code to Bitbucket. Learn how to generate a SSH key.

Add key

Obr. 8 Přidání SSH klíče
Zdroj: vlastní zpracování

Poté se zobrazí modální okno *(Obr. 9)*, kde uživatel vyplní název klíče, běžně se zde uvádí název počítače, se kterým je klíč svázán a samotný dříve zkopírovaný klíč. Poté je již uživatel považován za uživatele s oprávněným přístupem a Git mu vrátí hlášku o úspěšném pushi.



Obr. 9 Přidání SSH klíče, 2. krok
Zdroj: vlastní zpracování

Ohledně příkazu „push" je ještě důležité zmínit jeho výchozí nastavení, v němž je dobré pro každého uživatele změnit jednu důležitou hodnotu „push.default" na „current":

```
git config --global push.default current
```

Ve verzích Gitu 2.0 a nižších je totiž defaultně nastavená hodnota push.default na „matching", což znamená, že push nahraje do vzdáleného repozitáře všechny lokální větve, namísto té aktuální, což je právě nastavení „current". Nastavení „push.default" na „matching" má totiž 2 velké nevýhody, proto se ho nehodí mít nastavené *(SANTOS VELASCO, 2012)*.

1) Uživatelé si při tvorbě nové „featury" většinou zakládají „feature branche", které pak „mergují" (rebasují) do hlavní vývojové větve (většinou beta či master). Pokud však mají nastavený „push.default" na „matching" pošlou na „remote repository" i všechny jejich „feature branche", které tam dělají zbytečný nepořádek. Nejsou

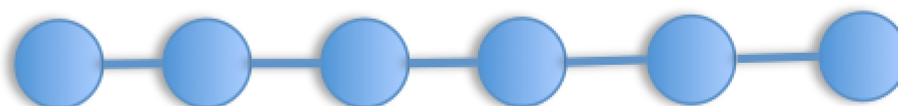
tam k ničemu, akorát znepřehledňují celý projekt, v horším případě nastane případ číslo 2.

2) Pokud nezkušení uživatelé „forcepushují“ a mají neaktuální větve, přepíšou ve vzdáleném repozitáři všechny větve a zničí tak cizí práci. V Gitu sice žádná úprava nemůže být ztracena, protože je uchována v historii prakticky navěky, ale nepříliš jednoduchá extrakce původní verze z přepsané historie rozhodně akorát zbytečně zabírá čas.

3.5.13 Týmová práce s gitem (větve)

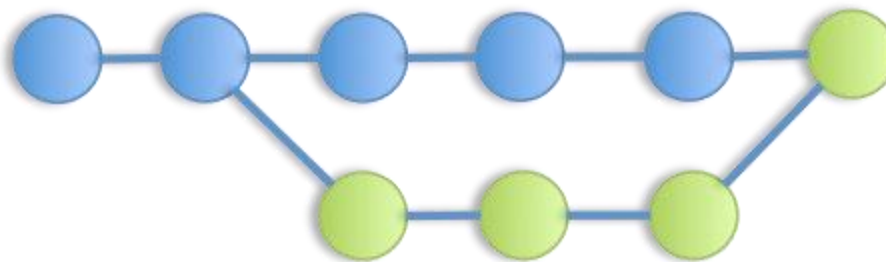
Doteď byla předvedena pouze práce jednoho uživatele, ale jak bylo řečeno již v úvodu, to kde Git vyniká nejvíce, je právě práce více lidí v týmu. K tomuto účelu využívá Git tzv. větve.

Větve jsou základním nástrojem, který umožní pracovat více vývojářů na stejných souborech, aniž by si je navzájem přepisovali. V momentě, kdy vývojáři potřebují ve stejnou dobu rozšiřovat projekt o další funkcionalitu, začnou větvit *(SWICEGOOD, 2010)*. Tento princip může být předveden na zjednodušeném příkladu asi takto: Existuje projekt, jehož vývoj doteď probíhal lineárním způsobem. Existuje v něm již několik desítek commitů a žádné z těchto commitů nebyly vytvářeny ve stejnou dobu, v Gitovské interpretaci historie by tak projekt mohl vypadat například takto:



Obr. 10 Lineární historie projektu
Zdroj: https://shane.io/images/posts/linear-vs-merge-based-histories.png

V tomto diagramu je jasně vidět, že se jedná o lineární historii, kdy mohl být každý další commit proveden až po jeho předchůdci. Jenže to je právě problém, pokud by takto chtěli pracovat na projektu 2 vývojáři, musel by jeden čekat na druhého, proto větve umožňují dostat se do stavu vypadajícího takto:



Obr. 11 Paralelní historie projektu
Zdroj: https://shane.io/images/posts/linear-vs-merge-based-histories.png

Čili modrý vývojář číslo 1 pracuje ve svojí modré větvi a zelený vývojář číslo 2 si v momentě, kdy potřebuje začít pracovat souběžně s vývojářem číslo 1, udělá kopii celého projektu, čili se vyvětví, pracuje tak na oddělených souborech a oba vývojáři si nemají šanci navzájem nic přepsat. Jakmile má zelený vývojář hotovou svou část práce, může ji začlenit zpět do modré větve vývojáře číslo 1 a nikdo nepřijde o svou práci. Git se o začlenění jedné větve do druhé postará automaticky a změny do sebe sloučí, jediné, na co upozorní, jsou právě části, kde došlo k úpravě od obou vývojářů, čemuž se říká konflikty. Tyto konflikty spolu oba vývojáři vyřeší a tím je hotovo. Žádný z vývojářů nepřišel o svou práci a mohl v klidu paralelně vyvíjet, aniž by mu někdo soustavně přepisoval jeho práci.

3.5.14 Příkazy pro práci s větvemi

V praxi slouží pro vytvoření nové větve příkaz *(Git Documentation, 2017)*:

```
git branch nazev_vetve
```

Pro přepnutí se do této větve slouží příkaz *(Git Documentation, 2017)*:

```
git checkout nazev_vetve
```

Pozn.: Aktuální větev může být v příkazové řádce viděna v závorce za složkou, ve které se nacházíme. Výpis příkazové řádky po zavolání výše uvedeného příkazu by tak mohl vypadat takto:

```
nazev_uzivatele@nazev_pc:/var/html/projekt (nazev_aktualni_vetve)$
```

Git bash ve Windows aktuální větev v závorce ukazuje v defaultním nastavení. V čistém Gitu, např. pokud používá uživatel Linux, však toto natavení chybí a je potřeba nastavit v souboru „.bashrc“ *(VAN AERLE, 2013)*.

Výše uvedené 2 příkazy pak jdou ještě spojit v jeden jako:

```
git checkout -b nazev_vetve
```

čímž je vytvořena větev s názvem „nazev_vetve“ a rovnou je do ní přepnuto. Názvy větví v projektu by měly být pojmenované nejlépe podle jednotné konvence. Dobrý způsob je např. první 2 písmena iniciály autora, pak podtržítko a následně k čemu větev slouží, a to vše nejlépe v angličtině. Název větve by pak mohl vypadat např. takto: pn_responsiveHeader, z čehož lze vyčíst, že větev vytvořil Pavel Novák (za předpokladu, že je ve firmě pouze jeden vývojář s danými iniciály) a v jeho větvi je vytvořená responsivní hlavička.

Nyní probíhá práce ve větvi úplně stejně jako bez větví. Uživatel vytváří změny, které následně commituje. Nová funkcionalita však přichází na řadu v momentě, kdy chce uživatel začlenit svou větev do větve jiné, většinou svou feature branch do hlavní vývojové větve.

Příkazy pro toto sjednocení existují 2. Jedná se o „merge“ a „rebase“ (CHACON, 2009).

3.5.14.1 Merge

Pomocí příkazu merge lze provést začlenění jedné větve do druhé následovně: První je potřeba přepnout se do větve, do které chceme začlenit druhou větev. K tomu opět poslouží příkaz:

```
git checkout master
```

V tomto případě má hlavní vývojová větev klasicky název „master“. Pro sjednocení této větve s větví „pn_responsiveHeader“ následně zavoláme příkaz:

```
git merge pn_responsiveHeader
```

Nyní můžou nastat 3 různé typy výstupů tohoto příkazu:

a) **Ve větvi master nedošlo od vyvětvení feature branche k žádnému dalšímu commitu**

   Toto je nejjednodušší případ, který může nastat, říká se mu také fast-forward (rychle kupředu), protože Git pouze přesune ukazatel vpřed na commit ve feature branchi. Není zde co slučovat, protože v paralelní větvi mezitím nevznikla žádná práce. Výpis příkazové řádky pak vypadá např. takto:

```
Updating f0958e1..655f002
Fast-forward
 main.css | 10 +++++++++-
```

```
 1 file changed, 9 insertions(+), 1 deletion(-)
```

b) Ve větvi master došlo od vyvětvení feature branche k dalším commitům, avšak ve slučovaných větvích nedošlo k žádným konfliktům (úpravám na stejném místě kódu)

Tento případ si lze představit např. tak, že uživatel číslo 1 v hlavní větvi upravil část kódu na začátku souboru a vývojář číslo 2 ve feature branchi naopak na konci souboru. Jejich změny tak Git může lehce tzv. auto-mergovat, čili do nového commitu automaticky sloučit a začlenit obě změny. Nový commit tak bude obsahovat soubor jak s úpravami od vývojáře 1 na začátku souboru, tak s úpravami od vývojáře číslo 2 na konci souboru. Výpis příkazové řádky pak může vypadat např. takto:

```
Auto-merging main.css
Merge made by the 'recursive' strategy.
 main.css | 2 ++
 1 file changed, 2 insertions(+)
```

Jak už bylo zmíněno, Git v tomto případě vytvoří commit. Jakmile totiž začleňuje změny z obou větví, nestačí pouze přesunout ukazatel aktuálního commitu, ale musí být vytvořen nový tzv. merge commit. Git v tomto případě po výpisu výše ještě otevře defaultní editor pro psaní commit message, kde musí uživatel provádějící commit napsat commit message s popisem, jaké změny byly mergovány a po jejich potvrzení teprve vznikne nový commit.

c) Ve větvi master došlo od vyvětvení feature branche k dalším commitům a ve slučovaných větvích došlo ke konfliktům v souborech.

Toto je nejsložitější případ ze všech mergů. Stejně jako v minulém případě byly prováděny úpravy paralelně v obou větvích, avšak od uživatelů pracujících v těchto větvích došlo k úpravě stejné části kódu.

Git není schopný z logiky věci tuto situaci vyřešit sám automaticky jako v předchozích 2 případech, a tak v této situaci přistupují na řadu samotní uživatelé, kteří musí konflikty ve stejných částech souboru vyřešit ručně. Git samozřejmě i v tomto případě funguje jako dobrý pomocník, protože označí přesně místa v souborech, ve kterých došlo ke konfliktům. Výpis příkazové řádky v tomto případě vypadá např. takto:

```
git merge pn_responsiveHeader
Auto-merging main.css
CONFLICT (content): Merge conflict in main.css
Automatic merge failed; fix conflicts and then commit the result.
```

Z tohoto zápisu lze vyčíst, že Git našel konflikty, které nedokáže sám vyřešit a radí uživatelům, nechť konflikty vyřeší manuálně a následně provedou merge commit s vyřešenými konflikty.

Uživatelé, jejichž soubory jsou v konfliktu, by se měli v této situaci kontaktovat a nalezené konflikty spolu vyřešit, tzn. rozebrat spolu, z jakého důvodu upravovali stejnou část kódu a která varianta bude lepší, případně jaké nové řešení vymyslí, aby byly zachovány funkcionality obou variant. Toto je velice důležitý krok. V praxi často totiž dochází k tomu, že konflikty řeší uživatel, který dělá merge a bez znalosti okolností, které vedli druhého uživatele ke změnám na stejném kódu, jeho změny odstraní a nahradí svými, případně nesprávně spojí obě změny, čímž změny druhého uživatele taktéž poškodí.

Samotné řešení konfliktů pak probíhá tak, že uživatel otevře soubory, ve kterých Git nahlásil konflikt a hledá v těchto souborech Gitem označené části kódu, kde došlo ke konfliktu. Konfliktní soubor by tak mohl vypadat např. takto:

```
h1{
color:red;
<<<<<<< HEAD
font-size:24px;
=======
font-size:20px;
>>>>>>> cssVetev
font-weight:500;
}

h2{
<<<<<<< HEAD
color:yellow;
```

```
font-size:15px;
=======
color:blue;
font-size:18px;
>>>>>>> cssVetev
fiont-weight:300;
}
```

Z toho lze vypozorovat gitovskou konvenci značení konfliktů. Začátek konfliktu je značen:

```
<<<<<<< HEAD
```

Po čemž je zobrazena úprava v aktuální větvi, oddělená:

```
=======
```

od úpravy v mergované větvi, jejíž konec je označen:

```
>>>>>>> nazevMergovaneVetve
```

Po odstranění těchto konfliktních symbolů, čili vybrání, zda zachovat variantu v jedné nebo v druhé větvi, lze provést merge commit:

```
git add.
git commit -m „Merge conglicts in main.css solved"
```

Zajímavé je ještě poukázat na to, že Git při fázi řešení konfliktů zobrazuje aktuální větev jako:

```
(master|MERGING)
```

A po vyřešení konfliktu již zpět jako:

```
(master)
```

což zvyšuje přehlednost, v jaké fázi se větve nacházejí.

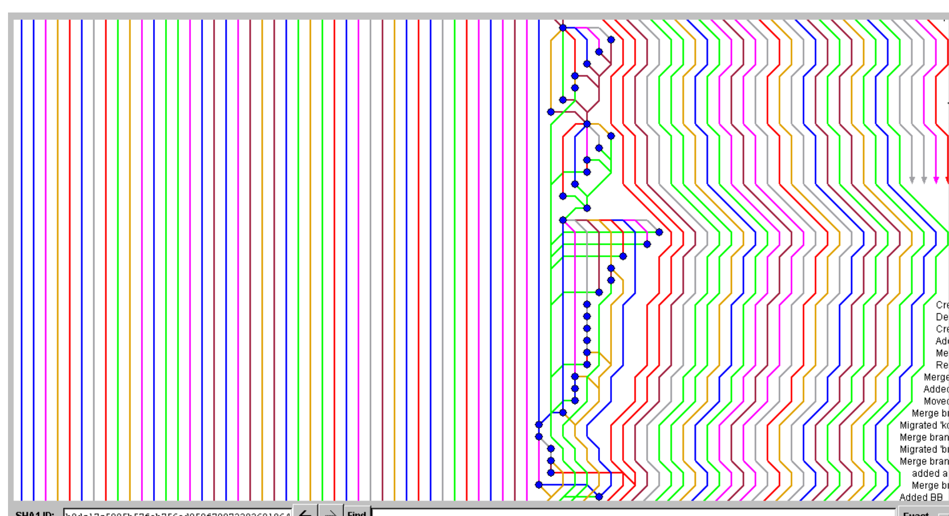
3.5.14.2 Rebase

Rebase je druhou možností, který příkaz použít, když uživatel potřebuje spojit 2 větve do jedné. Jedná se vlastně o velice podobný příkaz, který může vrátit stejné 3 stavy dle typu kolizí v souborech jako merge uvedený výše. Hlavní rozdíl je však v tom, že rebase oproti mergi nevytváří merge commit a zachovává lineární historii.

Lineární historie vzniká proto, že rebase proti mergi pracuje trochu odlišně, i když se stejným výsledkem. Při rebasi Git totiž vezme všechny změny od vyvětvení feature branche a aplikuje čili přeskládá (anglicky rebase) je na slučovanou větev. Ve slučované větvi je pak poslední commit posledním commitem z feature branche, který však v tu chvíli již zahrnuje i všechny předchozí změny z hlavní master větve. Nevzniká tak žádný merge commit ani rozvětvená historie, protože commity z vedlejších větví se jen přeskládají na hlavní větev a informace o původní větvi se ztratí. Zatímco při mergi se spojí 2 větve v jednu a obě zůstávají uložené v historii.

Pokud tak uživatel preferuje čistou historii, rozhodně bude volit sloučení větví přes rebase. Při slučování větví pomocí merge totiž lehce může dojít v historii při velkém počtu větví k něčemu, co se často nazývá „merge hell" a co je častým argumentem, proč merge nepoužívat a volit místo něj raději rebase.

Jak ukazuje *Obr. 12*, merge může být opravdové peklo:



Obr. 12 Merge hell
Zdroj: http://goats.con.com/images/merge-hell.gif

Samozřejmě merge má i výhodu, a sice že se díky merge commitu nikdy neztratí informace o tom, jak byl vyřešen konflikt. Pokud někdo vyřeší špatně konflikt při rebasi, informace o jejich řešení není nikde zaznamenaná.

4 PRAKTICKÁ ČÁST

4.1 PRÁCE S GITEM VE VÝVOJOVÉM PROSTŘEDÍ

Byť je používání Gitu v příkazové řádce rozhodně nejmocnějším nástrojem, tedy alespoň co se týče množství příkazů a kontrolou nad vstupy a výstupy, v praxi se často využívá grafických rozhraní, které integrují nejvíce používané příkazy na stisk pár tlačítek. Tato rozhraní mohou při základní práci s Gitem rozhodně urychlit práci a vnést přehled do stavu repozitáře díky perfektně zpracovaným grafických rozhraním. Dále pak skvěle slouží pro začátečníky nebo občasné uživatele Gitu, kterým tyto nástroje značně urychlí křivku učení a do jisté míry i zabrání tvorbě chyb. Pro využití všech možností Gitu, a hlavně využívání pokročilých funkcí však příkazová řádka nemá konkurenci, o to se snad ani grafická rozhraní nesnaží. Ve zjednodušení práce pro méně náročné uživatele však slouží výborně.

4.1.1 Vývojové prostředí Brackets

Jedním z vývojových prostředí, které podporuje práci s Gitem a na kterém bude tato práce demonstrována, nese název Brackets.

Brackets je vývojové prostředí vytvořeno společností Adobe Systems, stejné firmy která stojí za úspěšnými produkty jako Adobe Photoshop či Adobe Illustrator. Toto IDE čerpá inspiraci ze svého předka Adobe Dreamweaver, jehož vývoj byl ukončen roku 2011, aby ho nahradil právě Brackets, a to na konci roku 2014.

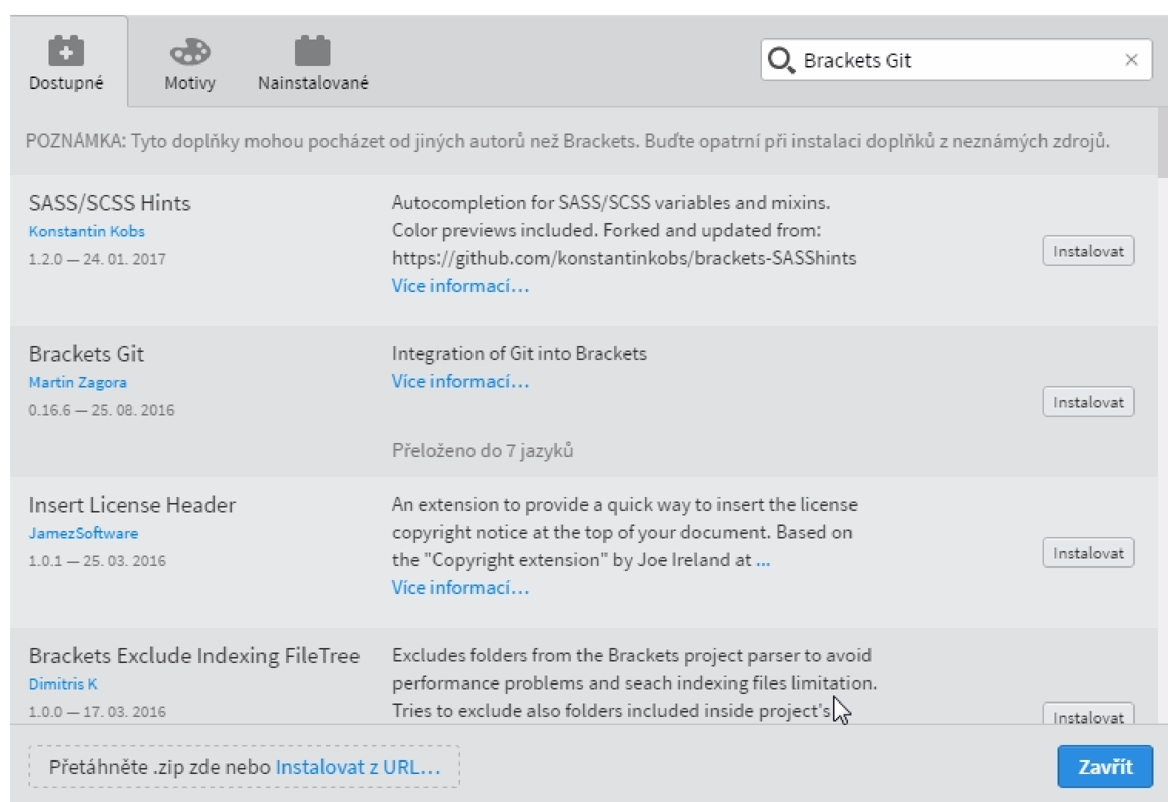
Brackets funguje pod všemi hlavními platformami, čili Windows, Linux i Mac, a to vše open-source pod MIT licencí.

Cílovou skupinou tohoto IDE jsou hlavně front-end vývojáři, jelikož díky svým nástrojům a funkcím umožňuje velice efektivní práci s HTML, CSS a JS. Hlavním lákadlem je pak integrace pluginu Adobe Extractor, který umožňuje snadnou práci s grafickými vrstvami vytvořenými v Photoshopu.

4.1.2 Plugin Brackets Git

Jelikož je Brackets open-source IDE vzniká do něj velké množství doplňků pro Git, nejznámější a nejvíce používaný je však Brackets Git, jehož hlavním vývojářem je Martin Zagora.

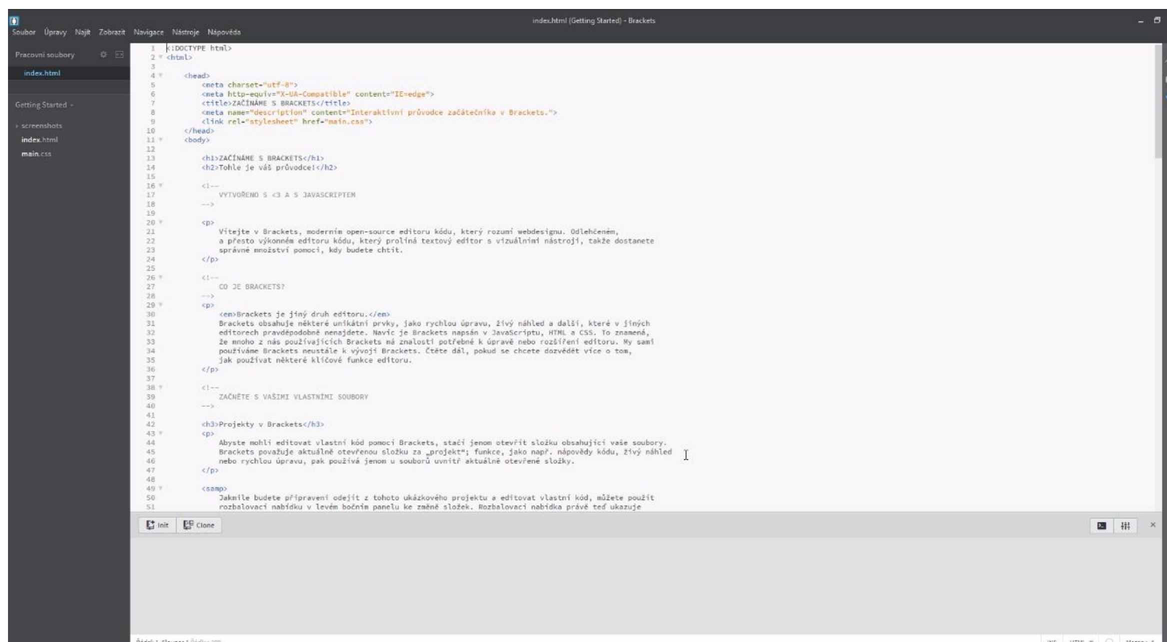
Brackets Git není součástí základní instalace Brackets, nýbrž funguje jako plugin. Jeho instalace však probíhá snadno, a to přímo v grafickém rozhraní tohoto editoru přes tlačítko v pravé liště „Správce doplňků", kde po vyhledání řetězce „Brackets Git" lze přes tlačítko „Instalovat" provést instalaci *(Obr. 13).*



Obr. 13 Instalace pluginu Brackets Git
Zdroj: vlastní zpracování

Následně je plugin spustitelný ze stejné pravé lišty jako se spouštěl výše uvedený „Správce doplňků". Plugin se následně otevře ve spodní části IDE.

Už na první pohled je vidět, že zde nebude probíhat žádná práce s příkazovou řádkou, ale vše bude otázkou klikání na tlačítka *(Obr. 14).*

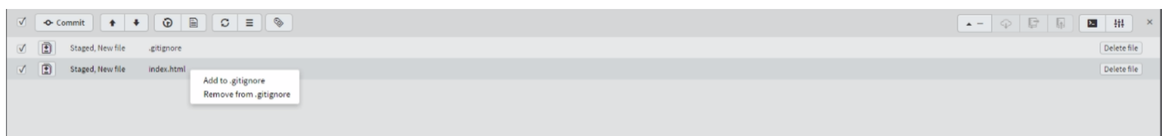


Obr. 14 Brackets s otevřeným pluginem Brackets Git
Zdroj: vlastní zpracování

4.1.3 Práce v Brackets Git

Brackets Git přizpůsobuje svoji nabídku akcí dle aktuálně otevřeného projektu (složky). Pokud je tedy otevřený projekt, kde ještě nebyl vytvořen repozitář, Brackets Git nabízí 2 hlavní tlačítka a sice „Init“ pro inicializaci nového repozitáře a „Clone“ pro naklonování již existujícího repozitáře.

Po zvolení jedné z těchto možností Brackets Git změní svůj vzhled dle nového kontextu, v tomto případě již existujícího repozitáře *(Obr. 15)*. V tomto zobrazení většinu prostoru zabírá graficky upravený výpis příkazu „git status“ tedy přehled, v jakém stavu se nacházejí soubory se změnami (untracked, staged). Untracked soubory pak lze přesunout do stage stavu pomocí zatržítka u daného souboru, což je funkcionalita příkazu „git add“. Kromě přidání souboru ke sledování ho však lze i odstranit a to buď přímo fyzicky tlačítkem „delete“ případně pouze přidáním do souboru .gitignore a to pravým klikem na daný soubor a zvolením možnosti „Add to .gitignore“.



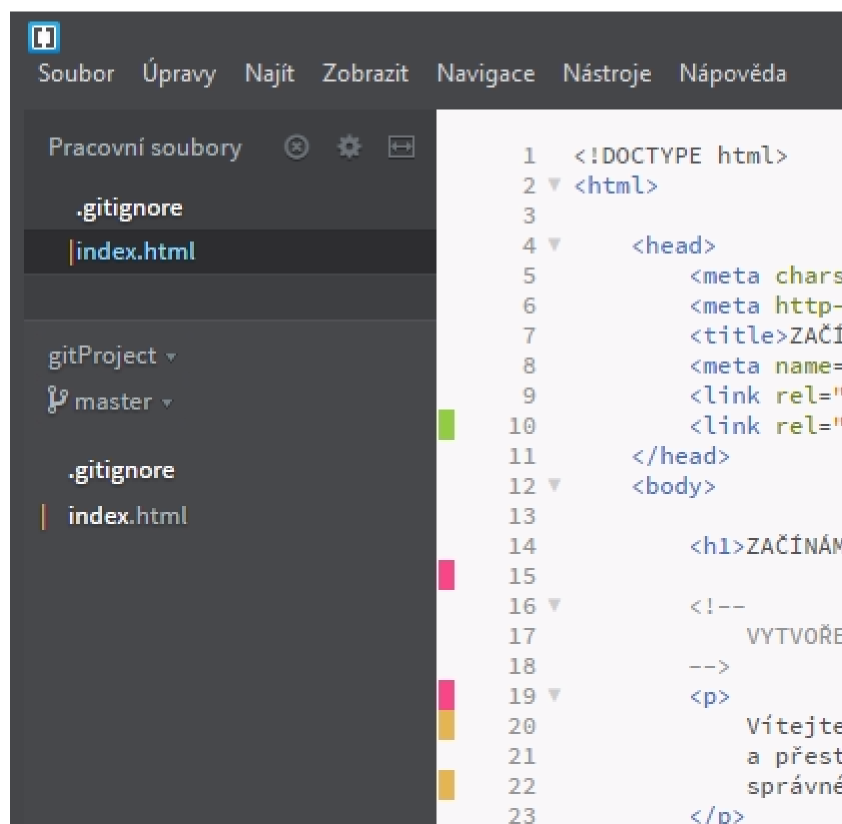
Obr. 15 Brackets Git v režimu existujícího repozitáře
Zdroj: vlastní zpracování

Dalším krokem po přidání změn do staging area je i zde stejně jako v příkazové řádce tlačítko commit pro stejnojmenný příkaz. Stisk tohoto tlačítka vyvolá modální okno, které zobrazí ve změněných souborech nově přidaný obsah zeleně a červeně obsah odebraný, takto upravený výpis by byl v příkazové řádce prakticky nedosažitelný. Uživatel si v tento moment může v klidu zrekapitulovat změny, které udělal, vše pořádně zkontrolovat, a nakonec pomocí textového pole přidat commit message a tlačítkem „OK“ potvrdit spuštění commit příkazu *(Obr. 16)*. Historii již vytvořených commitů pak lze sledovat přes ikonu hodin.

Obr. 16 Modální okno pro vytvoření commitu
Zdroj: vlastní zpracování

V režimu pluginu pro existující repozitář se však kromě hlavního okna pluginu mění vzhled i zbytku rozhraní Brackets. Stejně jako v okně pro vytvoření commitu, tak i v jednotlivých souborech, které Brackets edituje, jsou při každém uložení zvýrazněny změny na řádcích, a to dle stejné konvence, zelená barva pro nově přidaný obsah, červená pro odebraný a oranžová pro změněný.

Další změna nastane ve stromové struktuře otevřených souborů, kde dojde kromě stejné konvence barevného značení upravených souborů, hlavně k zobrazení aktuální větve a s tím spojené možnosti vytvářet větve nové, přepínat se do nich a větve slučovat (*Obr. 17)*.



Obr. 17 Brackets Git v režimu existujícího repozitáře 2
Zdroj: vlastní zpracování

Poslední z hlavních úkonů, které umí Brackets Git obsluhovat je nahrání lokálních změn na vzdálený repozitář a stahování změn z repozitáře zpět do lokálního prostředí. K tomu slouží tlačítka v hlavní liště pluginu příhodně nazvaná „Git push“ a „Git pull“. Pokud je uživatel zároveň zakladatelem repozitáře, má zde možnost i pro vložení adresy vzdáleného repozitáře přes tlačítko „Create new remote…“

Jako další spíše okrajové funkce pak Brackets Git nabízí tlačítka jako „Discard all changes since last commit“, „Undo last local commit“ či možnost změny údajů o uživateli v git configu (e-mail, jméno) a spoustu dalších nastavení a pokročilých funkcí nad rámec rozsahu této práce.

4.2 SOUČASNÉ MOŽNOSTI PRO PRÁCI S GITEM NA STRANĚ SERVERU

Tato část práce bude zaměřena na současnou situaci kolem podpory Gitu na straně serveru. Je totiž až s podivem, jak málo serverových řešení nabízí v dnešní době, 12 let od vzniku Gitu a přinejmenším 5 let od jeho nástupu jako nejpopulárnějšího verzovacího systému, alespoň částečnou podporu Gitu, natož pak všech jeho pokročilých možností. Samozřejmě existuje zde spoustu služeb s plnohodnotnou podporou Gitu, to nelze zpochybnit, problémem však je, že tato řešení jsou stále velice finančně náročná, konec konců si to mohou dovolit, pokud neexistuje dostatečná konkurence. Najít tak dobrý poměr cena/výkon na hostování aplikací se slušnou podporou Gitu je nelehký úkol.

Níže tedy bude popsáno několik řešení, která lze v současnosti použít pro práci s Gitem na serveru. Tato řešení budou následně mezi sebou srovnána, a nakonec navrhnuta nejlepší řešení, respektive více možných řešení dle potřeb jednotlivých uživatelů. Serverová řešení budou vybírána v kontextu hostování webových aplikací.

Ceny srovnávaných řešení budou převzaty z 5. 3. 2017, tzn. hodnota 25.44 Kč pro dolar, pro libru pak 31.28 Kč

4.2.1 Klasické české webhostingy

Zde je situace nejhorší ze všech případů, o to více je smutnější, že na těchto řešeních funguje většina internetu, přinejmenším toho českého, z čehož bohužel vyplývá ještě jedna nepříznivá informace a sice, že využití Gitu je i 12 let po jeho vzniku na velice špatné úrovni.

Mezi 5 největších z těchto hostingů patří v česku dle metodiky sdružení CZ.NIC *(Hositng - Statistiky CZ, 2017)* tito poskytovatelé:

1) **Wedos** (150 731)
2) **Forpsi** (96 834)
3) **Active24** (80 578)
4) **Czechia** (36 365)
5) **Český hosting** (32 284)

Pozn.: číslo v závorce udává počet domén hostovaných u těchto společností.

Z těchto TOP 5 českých hostingů však bohužel pouze poslední jmenovaný a sice „Český hosting“ nabízí podporu Gitu, a to je ještě otázkou, kolik z uživatelů tuto funkcionalitu používá, toto číslo však nebude moc optimistické.

Při porovnání cen, které tyto webhostingové společnosti požadují za 1 měsíc fungování služby, vychází tato čísla:

Wedos: 30,25 Kč

Forpsi: 24,20 Kč

Active24: 29,04 Kč

Czechia: 175 Kč

Český hosting: 100,83 Kč

Z těchto cen lze usoudit, že hosting s podporou Gitu vychází až 4x dráž než levnější hostingy bez jeho podpory, nicméně to je pouze situace u firem s největším podílem na trhu, u menších firem lze sehnat hosting s Gitem i za 30 Kč.

Co je ale víc zarážející, že existují i hostingy jako Czechia s nestandardní cenovou politikou a nevzhledným starým webem, které podporu Gitu nenabízí a stojí 175 Kč měsíčně. Zrovna tento hosting sice nabízí SSH přístup, otázkou je, zda jsou zde povoleny instalace vlastních služeb, ale vzhledem k tomu, že se jedná o jakýsi pseudovirtuální server, pravděpodobnost bude velice nízká. Navíc i kdyby instalace Gitu byla povolena, pořád musí uživatel nastavovat deployment skripty, které hostingy s defaultní podporou Gitu mají již přednastavené.

Z těchto informací lze pozorovat nepříznivý fakt a sice, že i největší jména na českém webhostingovém trhu zapomněla jít s dobou a Git ignorují. Problém však nemusí být nutně na straně hostingů. Ti se totiž řídí dle poptávky zákazníků a pokud zákazníci nejsou dostatečně kvalifikováni v oblasti vývoje softwaru a Git nepožadují, nepřecházejí ani ke konkurenci, která Git nabízí, a hostingové společnosti tak nemají důvod k jakýmkoliv upgradům svých serverů. Škoda tak jen, že nenabízejí Git alespoň v nějakém nadstandardním balíčku pro náročnější uživatele.

Naštěstí, jak už bylo zmíněno výše, je tu i spousta dalších společností, které s dobou jdou a Git nabízejí, bohužel však nejsou v popředí českého trhu. Jedná se např. o tyto firmy:

Název společnosti	Počet registrovaných domén:	Cena za měsíc hostingu:
Blueboard	9 561	95 Kč
eBola	6 891	47,19 Kč
MIXhosting	neuvedeno	30,25 Kč

Tab. 1 Ceny dalších hostingů s podporou Gitu
Zdroj: vlastní zpracování

Z této tabulky lze vyčíst, že lze sehnat webhosting s podporou Gitu prakticky za stejné peníze jako bez něj, takový MIXhosting je toho důkazem. Bohužel také krásný příklad toho, že pouze lepší technické řešení k získání zákazníků nemusí stačit a byť tu jsou firmy s horšími parametry svých plánů, prakticky bez technické podpory, stejně se díky dobře zvládnutému marketingu dokázali mnohem lépe prosadit.

4.2.1.1 Technické řešení

Důležité je samozřejmě zhodnotit tento typ služby nejen z hlediska dostupnosti a ceny, ale také technického řešení.

Podpora Gitu na těchto webhostinzích funguje následovně. Hosting má povolený SSH přístup a na serveru nainstalovaný Git. Na serveru tak může existovat vzdálený repozitář, do kterého jsou pushovány změny z lokálního repozitáře. Ale to není všechno. Tyto hostingy nabízí také automatický deployment. Na těchto serverech jsou totiž nastavené tzv. post-receive hooky, což je další vymoženost Gitu a které umožňují po pushi do vzdáleného repozitáře automaticky aplikovat změny v souborech do kořenového adresáře webové aplikace. Uživatel tak nemusí změny ručně nasazovat prostřednictvím zastaralého, nicméně bohužel stále velice populárního protokolu FTP.

Samozřejmě toto je i tak v zásadě to nejjednodušší využití Gitu na serveru. Toto řešení totiž trpí jedním zásadním nedostatkem. Nad repozitáři totiž není žádná grafická nadstavba na straně hostingu a nelze tak z webového rozhraní spravovat repozitář jako je tomu např. u Bitbucketu. Tyto hostingy umí přes webové rozhraní většinou základní věci jako správu SSH klíčů, ale tím jejich možnosti končí. Navíc pokud by chtěl uživatel nějak

customizovat chování repozitáře, nemá možnost, všechna nastavení jsou společná pro celý server, který má pod sebou několik hostingových účtů a úpravy na míru nejsou možné, navíc k nim nemá uživatel ani přístup.

Na druhou stranu pro základní použití je toto řešení víc než dostačující a při této ceně bezkonkurenční.

4.2.2 Ekosystém obyčejného webhostingu, repozitářového uložiště a deployovací služby

Tato možnost je takový kompromis, pokud uživatelé již používají některý z hostingů, který Git nepodporuje, ale přesto by ho chtěli začít používat, aniž by museli přecházet k jinému hostingu, který podporu Gitu má. Pro toto řešení mohou být různé důvody. Například pokud má uživatel, v tomto případě spíše firma, na současném hostingu již tolik aplikací, že by byl jejich transfer k jinému poskytovateli velice náročný a v konečném důsledku možná i ekonomicky nerozumný krok.

Postup povýšení současného bezgitovského hostingu na takové pseudogitovské, avšak funkční řešení probíhá takto. Uživatel si pro své aplikace prvně založí nějakou ze služeb repozitářových uložišť, např. Bitbucket nebo GitHub. Zde pro každou ze svých aplikací vytvoří repozitář. Tyto repozitáře však jsou však oddělené od hostingu. Pro jejich propojení existují různé deployovací služby jako např. DeployHQ či FTPLOY. Těmto službám dá uživatel přístup na FTP, kde má hostované svoje aplikace a přístup k repozitářovému uložišti, kde má uložené repozitáře, tyto služby pak dokáží tyto 2 segmenty spojit a samotná práce následně probíhá stejně jako na hostingu s podporou Gitu. Jakmile provede uživatel push změn do produkční větve, deployovací služba tuto změnu zaznamená a provede přesun změn z repozitáře na hosting. Jediný rozdíl v tomto případě je, že přesun změn neprobíhá přes SSH protokol, ale přes FTP protokol. Dalším rozdílem také je, že uživatel platí jak za hosting, tak za další 2 služby uložiště repozitářů a deployovací službu.

Ceny deployovacích služeb se liší podle počtu projektů a mohou se vyšplhat na celkem vysoké částky:

Free	Individual	Pro
Designed for everyone	Designed for freelancers	Designed for teams
One Project	Ten Projects	Unlimited Projects
Unlimited deployments	Unlimited Deployments	Unlimited Deployments
£0 forever	£6 per month *or* £60 per year	£12 per month *or* £120 per year

Obr. 18 Ceník plánů deployovací služby FTPLOY
Zdroj: vlastní zpracování

		Projects	Deployments
Free	It's Free	1	10 per day
Basic	£6.00/month 10 day free trial	10	Unlimited
Plus	£12.00/month 10 day free trial	22	Unlimited
Premium	£24.00/month 10 day free trial	48	Unlimited
Ultimate	£48.00/month 10 day free trial	100	Unlimited
Ultimate+	£90.00/month 10 day free trial	200	Unlimited

Obr. 19 Ceník plánů deployovací služby DeployHQ
Zdroj: vlastní zpracování

Z těchto ceníků lze vyčíst, že toto řešení se začíná vyplácet až od vyššího počtu projektů. Cena za 10 projektů je u obou služeb 6 £ což je cca 188 Kč korun, jeden projekt se tak prodraží měsíčně o 18.8 Kč. Pokud se však podíváme na opačnou stranu spektra, pokud by měl uživatel např. 200 projektů, platí průměrně 105 £ měsíčně, což vychází na cca 3284 Kč. Pokud však tuto částku podělíme počtem projektů, vychází měsíční cena na jeden projekt 16.42 Kč. To je o 14.5 % výhodnější řešení než při malém počtu projektů. Stále to je ale zdražení ceny hostingu o 60 %, pokud bereme za cenu hostingu průměr ceny 3. největších hostingů, což činí 27, 83 Kč.

Dále je potřeba platit repozitářová uložiště, jejichž cenová politika byla probrána v sekci „Repozitářová uložiště“ a která v nejlevnějším řešení stojí konstantních 10 £ (cca 313 Kč) měsíčně nezávisle na počtu projektů. To vychází při 10 projektech na 31.3 Kč na projekt měsíčně. Na druhé straně škály je však situace již o hodně zajímavější. Jelikož zde nedochází k žádnému zvýšení ceny za zvýšený počet projektů, cena při 200 projektech vychází na 1 projekt 1.57 Kč měsíčně, což je prakticky zanedbatelná částka.

Toto řešení se tak stává nejvhodnějším pro firmy, které chtějí přemigrovat na Git větší množství projektů, jelikož proti jednotlivci s málo projekty ušetří 14.5 % na deployment službě a celých 95% na repozitářovém uložišti. To však nic nemění na tom, že proti původní ceně hostingu došlo k nárůstu ceny o 17.99 Kč měsíčně. To sice není absolutně moc, relativně to je však stále cca o 65% více než za původní řešení bez Gitu.

Pro jednotlivce do 10 projektů pak vychází navýšení měsíční ceny o 50.1 Kč, což je už i absolutně celkem dost, relativně pak o to horších cca 180 % původní ceny navíc, což už naznačuje nevýhodnost.

Malé subjekty by tak měly raději přemigrovat pár projektů na hosting s integrovaným Gitem, který lze sehnat i cca stejně draze jako hosting bez podpory Gitu. Pro velké subjekty, je toto mnohdy nejlepší řešení, jak přemigrovat na Git a nemuset zároveň migrovat původní hosting s vysokým počtem projektů.

4.2.3 Vlastní server

Vlastní server je asi nejvíce odvážná varianta ze všech typů provozování Gitu, na druhou stranu nabízí bezkonkurenční správu a nemalé ušetření nákladů proti pronajatým serverům, pokud uživatel tyto servery provozuje dostatečně dlouho. Uživatel ho však prvně musí nakonfigurovat včetně Gitu, což zabere spoustu času. Dalším problémem je zajištění dostupnosti serveru, tzn. zajištění neustálého přístupu elektrického proudu a připojení k internetu. Toto spousta majitelů vlastních serverů obchází přes server housing, který je schopen tyto potřeby garantovat.

4.2.3.1 Technické řešení

Nastavení Gitu na vlastním serveru probíhá takto *(How To Set Up Automatic Deployment with Git with a VPS, 2013)*:

Nejdříve je potřeba vytvořit vzdálený repozitář a ten následně inicializovat:

```
mkdir site.git && cd site.git
git init -bare
```

Jakmile je repozitář inicializován, jsou v něm vytvořeny Git složky. Pro potřeby deploymentu je důležitá složka „hooks“. V této složce jsou umístěné skripty reagující na různé Gitovské akce, např. na push, což je potřeba právě u deploymentu. Hook reagující na push se nazývá „post-receive“.

```
cd hooks
cat > post-receive
```

Tento hook je pak potřeba napsat do podobné formy jako:

```
#!/bin/sh
git --work-tree=/var/www/domain.com --git-dir=/var/repo/site.git
checkout -f
```

První řádek pouze říká, pomocí čeho má být skript interpretován (linuxový bash). Důležitější je však druhý řádek. Ten se stará o všechnu práci stojící za deployem pushnutých změn z repozitáře na server. Pomocí tohoto příkazu je proveden checkout nebo-li přesun změn z repozitáře, jehož umístění je zapsáno v proměnné „--git-dir“ do kořenového adresáře aplikace, jehož cesta je zapsaná v proměnné „--work-tree“.

Jako poslední je potřeba tomuto hooku dát práva ke spouštění:

```
chmod +x post-receive
```

Nyní již je vše připraveno a při dalším pushi budou změny z repozitáře nahrány do adresáře aplikace.

Toto je tedy funkční základ, jak lze zprovoznit deploy změn z repozitáře na vlastním serveru. Samozřejmě dále by uživatel řešil problémy jako správu uživatelských oprávnění, nasazení webového rozhraní pro přístup k repozitářům, atd., nicméně to by vydalo na samostatnou práci.

4.2.4 Virtuální/dedikované servery

Virtuální a dedikované servery jsou další možností, kterou použít pro práci s Gitem. Ve spoustě aspektů jsou podobné klasickým hostingům, ve spoustě ale také odlišné.

Tento typ služby používají většinou firmy, které mají pod správou stovky webů a zřizovat pro každý web samostatný hosting je pro ně nepraktické i ekonomicky nevýhodné. Dle počtu projektů, které potřebují hostovat pak volí buď virtuální nebo dedikovaný server.

Dedikovaný server je služba, při které si uživatel pronajme celý server, na kterém běžně funguje třeba i několik set hostingů. Pokud však uživatel tak velkou kapacitu nevyužije, může pořídit právě virtuální server, což je virtualizovaná část dedikovaného serveru, která se tváří jako samostatný server, ve skutečnosti však není.

Důležité je upozornit na to, že až na pár výjimek nenabízejí poskytovatelé pro tento typ služby technickou podporu. Vše je zde v režii uživatele, který má na serveru volnou ruku a může zde vše instalovat a nastavovat dle jeho potřeb. A jedním z toho, co lze naistalovat je logicky právě i Git, který by na běžném hostingu bez jeho podpory poskytovatel služby na zákazníkovo přání nenainstaloval.

Tento typ služby tak volí většinou lidé, kteří mají zkušenosti se správnou vlastního serveru a přesně vědí, co dělají. Čistá instalace Gitu totiž sice umožní pracovat s repozitáři, ale třeba takový deployment už je potřeba manuálně nastavit, viz část „Vlastní server“.

Dalším potencionálním problémem u tohoto typu služby může být cena. Takto vypadá současná situace na trhu, pokud budou porovnány nabídky největších poskytovatelů v ČR.

Poskytovatel	Cena virtuálního serveru	Cena dedikovaného serveru
Wedos	121 Kč	787 Kč
Forpsi	30 Kč	544, 50 Kč
Active24	482,79 Kč	7 260 Kč
Czechia	351 Kč	2 408 Kč
Český hosting	103 Kč	4 235 Kč

Tab. 2 Ceny virtuálních a dedikovaných serverů u největších českých poskytovatelů Zdroj: vlastní zpracování

Z této tabulky lze vyčíst, že ceny virtuálních serverů jsou ještě relativně přijatelné, Forpsi dokonce nabízí virtuální server prakticky za cenu běžného hostingu. Dedikované servery jsou však s cenou již o řád jinde a pro běžného uživatele jsou nedosažitelné.

Důležité je také upozornit na to, že tato tabulka zachycuje pouze nejlevnější servery z nabídky společností pro ilustraci, jaká nejnižší investice je potřeba na zřízení

tohoto typu služby. Nabídka jednotlivých balíčků těchto služeb je natolik rozmanitá, že by přesáhla rozsah této práce. Jednotlivé hostingy se liší hlavně ve výkonu serveru (procesor, RAM), velikosti uložiště a jeho typu (SSD, HDD), případně povoleným měsíčním trafficem. Některé balíčky dokonce nabízí několik hodin práce technika měsíčně, tzv. managed servery.

4.2.5 Ekosystém repozitářového uložiště a cloudové platformy

Ekosystém repozitářového uložiště a cloudové platformy využívá propojení repozitářů z uložišť repozitářů typu GitHub a BitBucket s cloudovými platformami pro hostování aplikací jako např. Microsoft Azure, Heroku, Google Cloud či Amazon Web Services. Tyto služby patří mezi ty dražší, na druhou stranu nabízí nespočet výhod proti konkurenčním řešení.

4.2.5.1 Repozitářová uložiště

Mezi největší repozitářová uložistě současnosti patří BitBucket a GitHub. Obě tyto služby mají dokonale zpracované webové rozhraní pro správu repozitáře ze strany serveru. Ať už jde o správu uživatelských účtů přes grafické znázornění stavu repozitáře, po možnost sledovat změny v jednotlivých souborech.

Ceny těchto služeb vystihuje následující tabulka.

Paid Plans	Users	Private Repository	Price [USD]
GitHub Personal	1	Unlimited	7
GitHub Organisation	5	Unlimited	25
	200 + 5		25 + 1800
Bitbucket Growing Team (min10 - unlimited)	10	Unlimited	10
	Unlimited		200

Tab. 3 Porovnání cen GitHub vs BitBucket
Zdroj: vlastní zpracování

Z tabulky je jasně vidět, že výhodnější z těchto služeb je rozhodně Bitbucket, kde balíček s týmem do 10 uživatelů a neomezeným počtem repozitářů vyjde na pouhých 10$ měsíčně (cca 254 Kč). GitHub přitom požaduje za poloviční počet uživatelů 2,5násobnou cenu 25$ (cca 636 Kč) a ani ve vyšších balíčcích na tom není lépe. Při počtu 200 uživatelů vyjde GitHub na neuvěřitelných 1825$ měsíčně cca (46 427 Kč), zatímco Bitbucket na pouhých 200$ (cca 5088 Kč). GitHub přitom nenabízí žádnou výraznou konkurenční výhodu, obě služby jsou si velice podobné.

4.2.5.2 Cloudové platformy

Cloudové platformy v tomto ekosystému zajišťují prostor pro hostování projektů. Aplikace z repozitářových uložišť jsou schopny být díky integračním nástrojům těchto služeb deployovány na cloudovou platformu. Tyto platformy nejsou klasickými hostingy. Je to současná špička pro hostování aplikací. Klasické hostingy mají totiž v současné době 2 hlavní nedostatky, nepodporují moderní technologie a nejsou škálovatelné. Na druhou stranu jsou celkem drahé.

Porovnání cen těchto služeb by bylo nad rozsah této práce, jelikož jsou, jak již bylo zmíněno škálovatelné a nenabízí pevné plány, uživatel si tak může zkonfigurovat vlastní plán dle jeho potřeb. Cena za využití všech důležitých služeb, kvůli kterým by si uživatel vybral právě cloudovou platformu, se může však může pohybovat kolem 500$ (cca 12 720 Kč) měsíčně. Samozřejmě tu existují i plány zdarma na vyzkoušení či za několik $ s omezenou nabídkou služeb.

5 ZHODNOCENÍ VÝSLEDKŮ A DOPORUČENÍ

Podpora Gitu ze strany serveru dnes není rozhodně samozřejmostí. Z porovnání jednotlivých služeb pro hostování webových aplikací bylo zjištěno, že obyčejné hostingy jsou již nedostačující, naopak je potřeba zvolit nějaký z méně zažitých přístupů pro vývoj, který umožňuje plně využít potenciál Gitu.

Klasický hosting bez podpory Gitu snad ani není potřeba komentovat. Práce s Gitem zde není možná, a proto ho nelze doporučit žádnému uživateli Gitu.

Klasický hosting s podporou Gitu již disponuje základními funkcemi většinou včetně nastaveného deploymentu. Při jeho nízké ceně, tak lze tento typ služby doporučit hlavně uživatelům s nižším rozpočtem, kterým však pro základní práci bude sloužit dostatečně. Rozhodně nelze tento typ služby doporučit náročnějším uživatelům, kteří požadují možnost vlastní konfigurace.

Ekosystém obyčejného webhostingu, repozitářového uložiště a deployovací služby je dobrou variantou pro uživatele, kteří již mají spoustu projektů na klasických hostinzích bez podpory Gitu. Pro ně je toto nejjednodušší cesta, jak rozšířit stávající řešení o podporu Gitu s použitím co nejméně práce a financí. Pro uživatele, kteří začínají s novým projektem je toto řešení finančně nevýhodné a ti by měli volit klasický hosting s podporou Gitu.

Vlastní server lze doporučit zkušeným uživatelům, kteří mají bohaté zkušenosti jak se správou serverů, tak s konfigurací Gitu. Pro klienty, kde musí být garantována dostupnost, je lepší toto řešení raději nepoužívat nebo zkombinovat se službou server housingu.

Virtuální či dedikovaný server je výhodný pro uživatele s velkým počtem projektů, pro které se stává finančně výhodným proti klasickým hostingům, které se při velkém počtu projektů začínají prodražovat. Jako příjemný bonus zde mají vyšší někdy dokonce úplnou kontrolu nad serverem a garantovanou dostupnost poskytovatelem. V některým balíčcích dokonce pár hodin správy od serverových techniků. Pro jednotlivce s malým počtem nelze toto řešení doporučit pro vysokou cenu.

Ekosystém repozitářového uložiště a cloudové platformy je pak jasnou volbou pro nejnáročnější uživatele pracují s nejmodernějšími technologiemi, které běžné hostingy nenabízejí. Podmínkou je pak vyšší rozpočet. Za ten však dostanou možnost škálovat

službu dle potřeby a hlavně možnost maximálně využít všech možností Gitu na serveru ve výborném rozhraní díky napojení na repozitářová uložiště.

6 ZÁVĚR

Bakalářská práce popsala účel a důvod vzniku verzovacích systémů spolu s nejznámějšími zástupci a jejich historickým pozadím. Samostatná část pak byla věnována historii Gitu. Systémy byly následně porovnány z hlediska hlavních charakteristik, hlavní pozornost byla věnována rozdílu centralizovaného a decentralizovaného systému. Práce s verzovacími systémy byla dále demonstrována na verzovacím systému Git. Na tomto systému byly popsány základní a nejdůležitější příkazy pro práci. Pozornost byla věnována hlavně instalaci Gitu, jeho konfiguraci, inicializaci vlastního či stáhnutí cizího repozitáře a v neposlední řadě přesunu změn od uživatele na server. Samostatná část pak byla věnována serverové službě BitBucket. Teoretická část byla uzavřena informacemi o práci v týmu, čili práci s větvemi, kde byly popsány 2 hlavní příkazy pro slučování změn a sice merge a rebase.

Praktická část pak názorně předvedla práci s Gitem ve vývojovém prostředí. Pro tyto účely bylo vybráno vývojové prostředí Brackets s pluginem Brackets Git. Toto lokální použití Gitu bylo následně obohaceno také o praktické možnosti nasazení Git na serveru. Tyto možnosti byly popsány v kontextu vývoje webových aplikací. Každá z možností měla popsána svůj princip, výhody a nevýhody, cenovou hladinu, případně rozdíly proti konkurenčním řešení. Nakonec byla dána doporučení, pro které uživatele jsou jaká řešení nejvhodnější a proč.

Bakalářská práce tak nejen shrnula teoretická východiska, ale díky praktické části i dala přínos v usnadnění rozhodování uživatelům hledající serverová řešení pro Git. Oba hlavní cíle bakalářské práce tak byly splněny.

7 SEZNAM POUŽITÝCH ZDROJŮ

Tištěné dokumenty:

CHACON, Scott. Pro Git [online]. Praha: CZ.NIC, 2009. CZ.NIC. ISBN 978-80-904248-1-4.

O'SULLIVAN, Bryan. Mercurial: the definitive guide. Sebastopol, CA: O'Reilly Media, 2009. ISBN 978-0-596-80067-3.

MASON, Mike. Pragmatic guide to Subversion. Raleigh, North Carolina: Pragmatic Bookshelf, 2010. Pragmatic programmers. ISBN 978-1-93435-661-6.

SOMASUNDARAM, Ravishankar. Git: version control for everyone beginner's guide: the non-coder's guide to everyday version control for increased efficiency and productivity. Online-Ausg. Birmingham, UK: Packt Pub, 2013. ISBN 9781849517522.

SWICEGOOD, Travis. Pragmatic guide to Git. Raleigh, N.C.: Pragmatic Bookshelf, 2010. Pragmatic programmers. ISBN 978-1-93435-672-2.

Elektronické dokumenty:

PRSKAVEC, Ladislav. Průřez historií verzovacích systémů [online]. 2011. Dostupné z: http://blog.prskavec.net/2011/10/prurez-historii-verzovacich-systemu/

TRENZ, Oldřich a Jan KOLOMAZNÍK. Softwarové inženýrství 1 [online]. 2011 . Dostupné z: http://is.mendelu.cz/eknihovna/opory/index.pl?cast=29397;fit_w=1;close_menu=1

Git Official Website [online]. 2017. Dostupné z: https://git-scm.com/

GitSvnComparison [online]. 2013 . Dostupné z: https://git.wiki.kernel.org/index.php/GitSvnComparison

LOURDAS, Vasilis. Git vs. Subversion: A performance comparison [online]. Dostupné z: https://www.lourdas.eu/blog/git-vs-subversion-performance-comparison

GitHub Help [online]. 2017. Dostupné z: https://help.github.com/

GitHub Help: Authenticating to GitHub: Generating a new SSH key and adding it to the ssh-agent [online]. 2017. Dostupné z: https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/

How is git commit sha1 formed [online]. 2012. Dostupné z: https://gist.github.com/masak/2415865

SANTOS VELASCO, Antonio. Make git push the current branch by default [online]. 2012 [cit. 2017-03-09]. Dostupné z: http://blog.santosvelasco.com/2012/07/15/make-git-push-the-current-branch-by-default/

Git for Windows [online]. 2017. Dostupné z: https://git-for-windows.github.io/

VAN AERLE, René. Git tip: Show your branch name on the Linux prompt [online]. 2013. Dostupné z: https://www.leaseweb.com/labs/2013/08/git-tip-show-your-branch-name-on-the-linux-prompt/

Hositng - Statistiky CZ [online]. 2017. Dostupné z: https://stats.nic.cz/stats/hosting/

How To Set Up Automatic Deployment with Git with a VPS [online]. 2013. Dostupné z: https://www.digitalocean.com/community/tutorials/how-to-set-up-automatic-deployment-with-git-with-a-vps

Git Documentation [online]. 2017 Dostupné z: https://git-scm.com/docs

MCMILLAN, Robert. After controversy, Torvalds begins work on "git". PCWorld [online]. 2005. Dostupné z: http://www.pcworld.idg.com.au/article/129776/after_controversy_torvalds_begins_work_git_/