Verzovací systémy

Verzovací systémy

Verzovací systémy jsou nějaký systém či software, který je určen pro správu změn jakýkoliv projektů a prací. Projekt se často v průběhu času mění a to nejvíce ve vývoji, kdy jsou klíčové části vytvořeny.

Verzovací systémy často mohou být i jen nějaké postupy a doporučení, kterými se jednotlivci i týmy řídi. Verzovací systémy se nejčastěji používají právě pro práci mezi více lidmi, aby byl ve změnách přehled a mohlo pracovat nezávisle na sobě. Klasické kopírování složek (např. s jménem datumu změny) je do jisté míry taktéž verzovací systém, ten nás ale nechrání vůbec proti ztrátě dat.

Název revize se ve verzovacích systémech používá pro označení balíčků změn. Například revize je změnění několika souborů v projektu. Často se zaměňuje s názvem verze.

Verzovací systémy jsou často i v jistých programech a webech. Jako příklad můžeme uvést Google Docs, které ukládají historii verzí dokumentu, ve kterých můžeme hledat a dokonce se i k verzi vrátit.

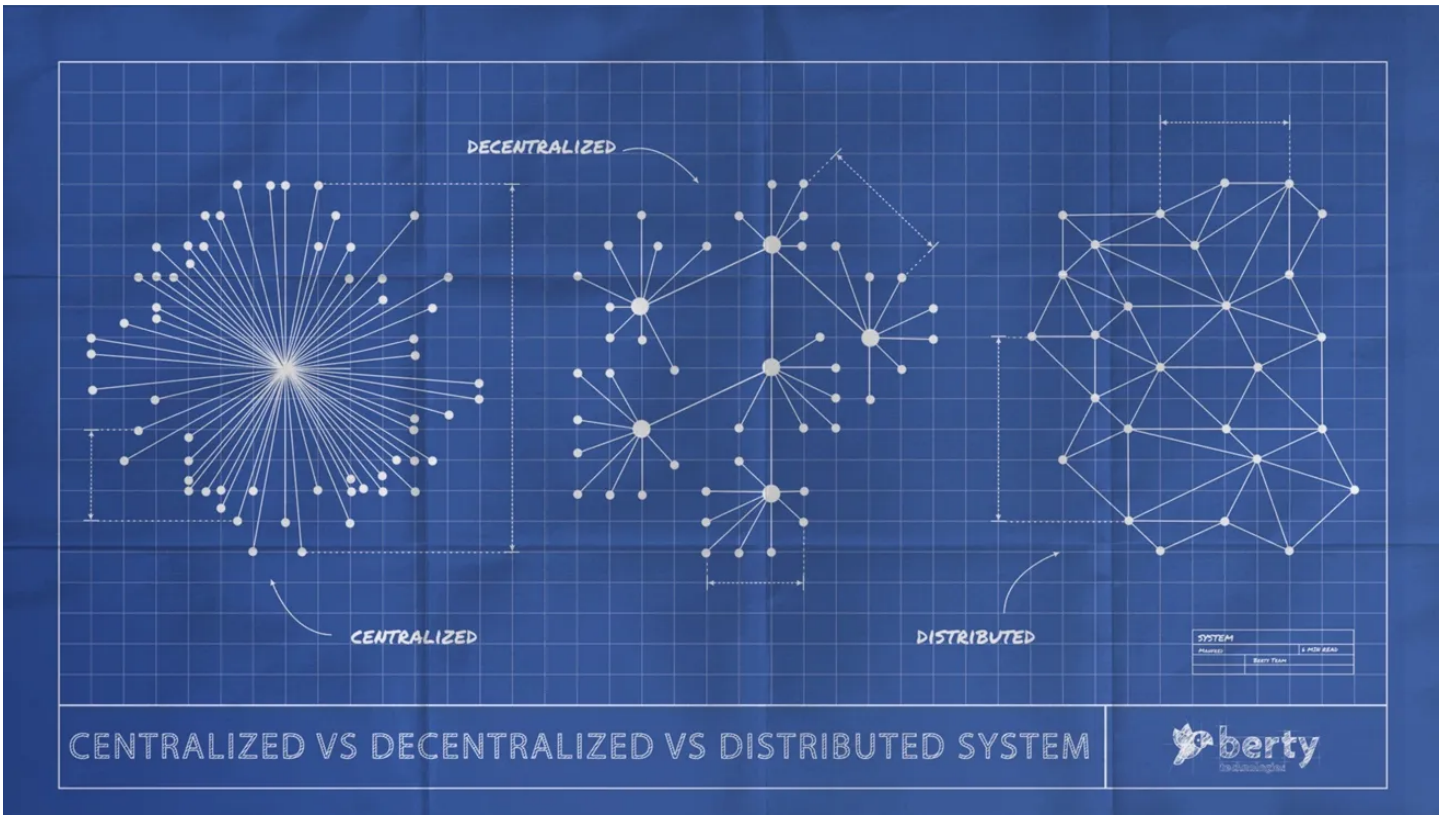
Verzovací systémy se používají na řadu typů projektů, nejčastěji však na ty spojené s IT (kódy, nastavení, weby, ...) ale lze je používat i na grafické či vlastně jakékoliv jiné. Práce s binárními soubory (např. obrázky) je však komplikovanější. Verzování nám pomůže všude, je totiž jednoduché se dívat na historii a případně se k ni vracet. Sjednocení změn z různých zdrojů (při práci více lidí najednou) se taktéž zjednoduší.

Práce se soubory

Verzovací systémy často používají různé styly práce se soubory. Jeden z nich je zamykání souborů a spojování změn. Zamykání souborů je způsob, kdy se celý projekt nebo soubor před prací jedince zamkne. V tomto stavu nemůže nikdo jiný na souboru pracovat, resp. jej uložit (max. číst). Tím zcela zmizí problém sjednocování změn (více lidí pracovalo, teď je musíme dát dohromady), ale zase může pracovat pouze jeden člověk na souboru (v případě projektu jen jeden zároveň). To je pro rychlé vytváření projektů velmi neefektivní. Často se totiž stává, že člověk může nechat soubor nechtěně zamknutý. Na tento styl se hodí právě grafické soubory a další binární soubory.

Slučování změn je mírnější. Na projektu může pracovat kdokoliv a kdykoliv, jen když se soubor uloží a někdo jej předtím ještě změnil, musí se určit, jak ten soubor vlastně má vypadat — sloučit se do pravého stavu. Na tento typ se hodí jakýkoliv typ souboru.

Umístění verzování



Centralizované

Existuje pouze jeden autoritativní server (umístění), kde jsou všechny změny uchovány nehledě na uživatele.

Decentralizované

Mají různé majitele, na které se pojí další uživatelé, nemusí vědět o sobě navzájem.

Distribuované

Všichni vědi o ostatních a všichni mají ke všem verzím přístup.

Toto umístění nemusí automaticky znamenat, že daný typ automaticky komunikuje se serverem. Velmi často se komunikuje jen tehdy, když je to potřeba.

Historie verzí

Obecně existují pouze dva typy historií verzí ve verzovacích systémech. Jedná se o lineární a grafovou historii. Lineární je jednoduchá — jedná se o list, ve kterém každá verze ukazuje na tu předchozí a první neukazuje na nic. S tímhle typem se moc nesetkáme u týmových projektů.

Grafová historie — jedná se o orientovaný (tj. ukazuje na něco) acyklický (tj. nemá cyklus, nelze se postupem vpřed dostat do nějaké předchozí verze) graf, kde z jedné verze může existovat více verzí, které se mohou různě spojovat.

Verzovací systémy

Známé verzovací systémy jsou:

- Git

- Subversion
- CVS
- Bazaar

Úvod Gitu

Git je opensource (tj. můžete se podívat na kód) verzovací systém, který je velmi populární. Používá ho řada služeb a programátorů a v podstatě nemá v moderním světě žádnou velkou konkurenci. Byl vytvořen v roce 2005 Linusem Torvaldsem a Juniem Hamanem. Používá řadu protokolů, jako jsou HTTP, SSH* a i FTP.

Cíle Gitu jsou:

- být jednoduchý (pro naučení stačí pár hodin samostudia)
- podporovat distribuované použití (celá historie je u všech zařízení, mohou se stahovat a upravovat bez nutnosti serveru)
- být zdarma
- být rychlý (proto vlastní Git vzniknul, protože dřívější systémy pro vývoj jádra Linux byly velmi pomalé a některé akce trvaly i hodiny)
- ochránit proti ztrátě dat (dost aktivit v Gitu lze vzít zpět (resp. získat data zpátky) a již z principu je možné se vrátit do předchozích verzí)

Charakteristiky Gitu jsou:

- data se po odeslání jen tak neztratí (existují lokální kopie u klientů)
- změny se na sebe váží a jsou propojeny
- veškeré objekty v Gitu jsou reprezentovány jednoznačným identifikátorem pomocí hashovací funkce SHA-1
- podporuje různé OS, systémy a jazyky
- rychlost Gitu závisí především na hardware počítače a se sítí komunikuje pouze pokud to uživatel chce.

Velmi důležitou vlastností Gitu je to, že neukládá změny jednotlivých souborů, ale jejích snímky. Tedy, jak v určité verzi celý soubor vypadal. A pokud se soubor nezměnil, ukazuje to na nějakou předchozí verzi.

Git se rozděluje do tří částí, se kterými se pracuje:

- working directory, neboli pracovní adresář, což je složka ve které pracujete na projektu
- staging area, neboli připravené změny, což je speciální místo, do kterého dáváte soubory, které půjdou do revize
- git directory, neboli Git složka, ve které se nacházejí informace i Gitu a my je můžeme měnit

Pro práci s Gitem je nutné mít Git jako příkaz v příkazové řádce (tj. CLI). Prozatím budeme využívat právě příkazovou řádku (např. na Windows Powershell) a později si ukážeme Git klienty, které mají GUI.

Git má skvělou knížku, která celou práci s Gitem a jeho vnitřnosti popisuje a vysvětluje. Při čtení této lekce ji doporučuji číst ZDE — je skoro celá v českém jazyku!

---

* při použít SSH je nutné na danou platformu (případě server) nahrát svůj klíč a nebo dokonce mít vytvořen účet

Nastavení a nápověda Gitu

Jako první příkaz si ukážeme příkaz k nastavení gitu. Tento příkaz slouží především na nastavení repozitáře (více později) ale i globálních nastavení. Nastavení používá páry klíče a hodnoty. Seznam všech klíčů naleznete ZDE.

Jako základní nastavení po nainstalování by se hodilo nastavit správné jméno a e-mail, který se používá skoro všude jako reference na to, kdo změny tvoří.

```
git config --global user.name "Pepa Kopecky"
git config --global user.email kopecky.pe.2020@skola.ssps.cz
```

Výše uvedený příkaz nastaví globálně (dle přepínače `--global`) klíče `user.name` a `user.email` na dané hodnoty. Je-li hodnota víceslovná, je nutné celou hodnotu obalit uvozovkami.

Můžeme dokonce vypsat veškerá nastavení a určité nastavení dle klíče.

```
git config --list
git config user.name
```

Je důležité si uvědomit, že nastavení může být vázáno na repozitář (tj. bez `--global`). Vždy má prioritu nastavení repozitáře a až potom globální.

Všechny příkazy lze vyvolávat v jakékoliv složce repozitáře. Příkazy vždy najdou nejbližší rodičovskou složku, která obsahuje složku `.git`.

Pokud si nebudete s nějakým příkazem vědět rady, je možné si pro něj zobrazit přímo pomocí příkazu nápovědu.

```
git help <příkaz>
git <příkaz> --help
git help config
```

Repozitář

Repozitář je název pro projekt, který je verzovaný pomocí Gitu. Jedná se tedy o složku, ve které je projekt, který je verzován. Tento repozitář musí být vytvořen (např. již v založeném projektu) a nebo to děláme ještě předtím. Často repozitář taktéž tzv. klonujeme, tedy stáhneme z internetu, ale to si povíme až v dalších kapitolách.

Repozitář má velmi důležitý úkol, a to je, sledovat veškeré změny projektu. Uvnitř této složky nalezneme taktéž složku `.git`, která uchovává veškerou historii, nastavení a cokoliv co Git potřebuje.

Založení repozitáře

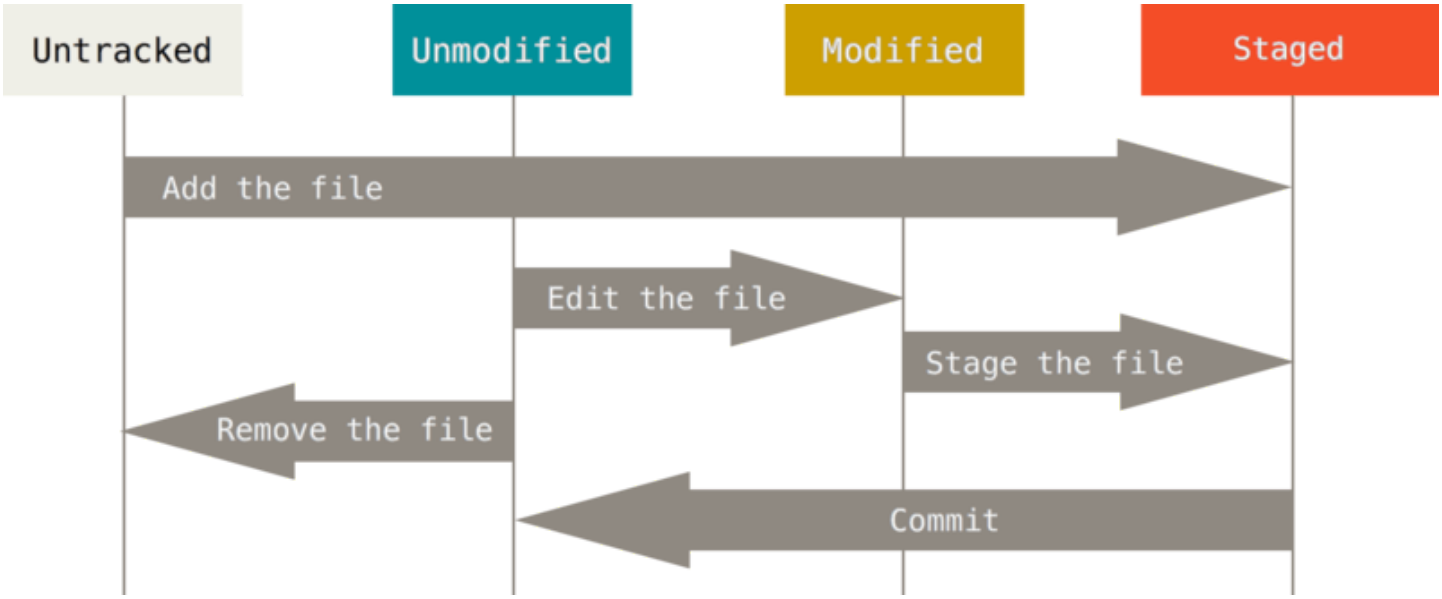
Založit repozitář nyní můžeme lokálně pomocí příkazu `git init`. Tento příkaz vytvoří složku `.git`, veškeré základní nastavení a ihned začne sledovat soubory. Když tuto složku smažeme, přijdete o veškerou historii projektu (prozatím lokální kopii).

Commit

Commit je balíček změn (již víme, že se jedná o snímky), které jsou již zaznamenané a ukončené. Jedná se tedy už o nějakou revizi (verzi) projektu, ve které se něco změnilo. Celý repozitář se z těchto commitů skládá (některé repozitáře jich mají miliony). Commit je vždy identifikován SHA-1 hashem, kterým na něj referujeme. Commit má taktéž svojí zprávu, časovou značku a autora. Většina commitů má alespoň jednoho rodiče (ze kterého své změny putují). Jen jeden commit nemá žádného rodiče a to je první commit v linii historie.

Git používá stavy souborů:

- untracked — soubor není sledován, jedná se o nový soubor, který v historii není
- unmodified — soubor již byl sledován, ale svůj obsah nezměnil
- modified — soubor je sledován a změnil se od poslední revize
- staged — soubor je sledován a je připraven být zabalen do commitu



Pro zjištění, v jakém stavu je aktuálně repozitář, lze použít příkaz `git status`. Tento příkaz nám vypíše, stavy jednotlivých souborů projektu. Vynechává soubory, které se nezměnili. Ukázka výpisu:

```
On branch main
Your branch is up to date with 'origin/main'.


Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   5.html
```

```
        modified:   9.html


no changes added to commit (use "git add" and/or "git commit -a")
```

Přidávání souborů

Pokud chceme označit soubory jako staged, můžeme to udělat (jak nám říká nápověda výpisu statusu) pomocí příkazu `git add`. Lze například udělat:

```
git add <soubor>
git add README.md
git add path/to/another/file.txt
```

V případě, že označíme soubor jako staged, tak se označí na odeslání AKTUÁLNÍ stav soubor (snímek) a při další úpravě tyto úpravy nebudou automaticky jako staged označeny.

Existují taktéž různé přepínače a zápisy, které označí více souborů najednou:

`git add -A` — označí všechny soubor v celém repozitáři

`git add .` — označí všechny soubor v aktuální složce a v podsložkách (a rekurzivně)

`git add -u` — označí soubory, které byly modifikované a smazané, ale neoznačí nové soubory

---

Přesné změny revize

Příkaz git status ale není moc vhodný na to, abyste zjistili co se přesně změnilo (řádky kódu, ...). Proto existuje příkaz `git diff` a `git diff --staged`, kde první zobrazí ve formátu patch změny všech souborů ve working directory s minulou verzí. Druhý příkaz toto porovnání vytvoří pouze se soubory, které jsou staged.

Patch je formát, ve kterém se vám zobrazí jaké řádky byly přidány a smazány společně s nějakými metadaty (jaké commity, jaké SHA-1 hashe, ...). Plus na začátku řádky značí přidaný řádek a mínus smazaný řádek.

Zapsání změn

Nyní víme co je to commit, jak označit soubory jako staged ale nevíme, jak vůbec takový commit vytvořit. Po tom, co přidáme všechny námi chtěné soubory (resp. změny a jejich snímky), můžeme zavolat příkaz:

```
git commit -m "Main page redirect fix"
git commit
```

První řádek okamžitě vytvoří commit s uvedenou zprávou (o tom, jak by zpráva měla vypadat, si povíme v jiné kapitole). Druhý příkaz je komplikovanější. Ten vám otevře vámi nastavený editor (nebo základní pro úpravu souborů), ve kterém uvidíte všechny změny v commitu a do něj napíšete danou zprávu. Po uzavření souboru se commit vytvoří a zapíše.

Po zapsání změn se staged soubory odznačí a celý cyklus může začít znovu.

Speciální soubory gitu

Mimo složku `.git` umí Git pracovat i se soubory, které více specifikují práci. Jeden z těchto souborů je soubor `.gitignore`, který lze vložit do jakékoliv složky, ale nejčastěji se vkládá do kořenového adresáře. Tento soubor ukládá informace o souborech, které má Git ignorovat (nebude je sledovat a přidávat).

Soubor obsahuje různá pravidla, lze v něm používat různé zástupné znaky a negace. Níže je ukázka s pravidly a vysvětlením toho, co ignorují.

```
# Ignoruj všechny soubory (v jakékoliv složce), které končí na .log.
*.log

# Negace pravidla - přestože výše zakazuje tento soubor, sleduj ho.
!dev.log

# Ignoruj složku build v hlavním adresáři.
/build/

# Ignoruj soubor build v hlavním adresáři.
/build

# Ignoruj všechny složky lib (v jakékoliv složce).
lib/

# Ignoruj soubor config.xml v jakékoliv podsložce složky config.
/config/**/config.xml

# Ignoruj všechny soubory, které začínají config.
config*
```

Historie

Jeden z cílů verzovacích systémů je dovolit uživatelům sledovat, co se v daném projektu stalo. Nyní si ukážeme příkaz, který právě to dělá. Tento příkaz má spoustu přepínačů a každý je důležitější než ten předchozí. Neukážeme si taktéž všechny, ale ukážeme si ty nejdůležitější. Přepínače lze kombinovat a tím zobrazení vylepšovat.

V každém případě je výpis commitů v historii vždy chronologický a velmi často platí, že první záznam, který uvidíte, je nejnovější commit. Zobrazení v tomto příkazu je často zobrazováno v menším okně, ve kterém se lze pohybovat pomocí šipek a tlačítek PAGE UP a PAGE DOWN. Z tohoto okna odejdete pomocí zápisu `q`.

V základu je příkaz `git log` velmi strohý. Vypíše nám seznam commitů a neukáže nám žádné vazby (resp. větve, o kterých si povíme později). Bez přepínačů se nám ke každému commitu ukáže identifikátor, autor, časová značka a jeho zpráva.

Určení počtu výsledků

Pomocí přepínače `-x`, kde `x` (např. `-1`, `-30`) je celé kladné číslo můžeme určit počet commitů, které se nám v tomto zobrazení ukáže.

Statistiky souborů a commitu

Pomocí přepínače `--stat` můžeme ke každému commitu zobrazit jednoduchou statistiku, která říká kolik bytů, řádků a změn bylo vytvořeno pro daný soubor a commit. Ukázkový výpis:

```
commit 6cf7ae812af9af919f7ed22d2f0a12 (HEAD -> main, origin/main)
Author: Matěj Cajthaml <matej.cajthaml@ssps.cz>
Date:   Wed Jul 27 17:43:54 2022 +0200

    Automatic grading for works

 server/package-lock.json                          | 1528 ++++++++++++++++++-
 server/package.json                               |    4 +
 server/src/index.ts                               |    4 +-
 3 files changed, 1518 insertions(+), 18 deletions(-)
```

Jednořádkové zobrazení

Přepínačem `--graph` se zobrazí celá historie jako graf, na kterém můžeme vidět různé větve a jejich spojování. Větve jsou vždy po levé straně.

Omezení data a času

Pomocí přepínačů `--since`, `--before`, `--after` a dalších můžeme omezit, jaké commity se nám zobrazí. Pro datum používáme formát `YYYY-MM-DD` nebo zápisy jako `x.weeks` a obdoby. Ukázky:

```
git log --since=2.weeks
git log --since=2.days
git log --before="2022-09-01"
git log --after="2022-09-01"
git log --after="2022-09-01" --before="2022-10-01" --graph
```

Vyhledání dle autora či textu

Pomocí přepínačů `--autor` a `--grep` můžeme vyhledávat v autorech comittů a jejich zpráv. Ukázky:

```
git log --author="Matej Cajthaml"
git log --grep="redirect"
git log --grep="remove"
```

Návrat do předchozího stavu

Často se nám stává, že se musíme vrátit do předchozího stavu souboru či commitu. Nyní si ukážeme nejčastější akce.

Resetování aktuálně změněného souboru

Používá se pro případ, že chceme smazat veškeré změny, které se v daném souboru staly od minulé revize.

```
git checkout -- <soubor>
git checkout -- README.md
git restore <soubor>
git restore README.md
```

Odstranění souboru z označení staged

Odstraníme soubor z aktuálního stavu staged area. Nesmažou se změny, jen se odpřidá soubor. Příkazy:

```
git reset HEAD <soubor>
git reset HEAD todo.txt
git restore --staged <soubor>
git restore --staged todo.txt
```

Připojení se k minulému commitu

Používáme právě tehdy, když jsme vytvořili commit, ve kterém jsme zapomněli něco udělat, nebo jsme mu například napsali špatnou zprávu.

Před použitím příkazu přidáme do staged area věci, které chceme do minulého commitu přidat. Pokud nic přidat nechceme, nic nemusíme označovat. Poté zadáme příkaz `git commit --amend`, který poslední commit upraví. **V žádném případě se nevytváří nový commit.** Po odeslání příkazu se nám otevře editor pro změnu zprávy commitu. Zprávu můžeme přidat rovnou při volání příkazu pomocí přepínače `-m`, který je i u příkazu `commit`.

Návrat ke commitu

Často se potřebujeme dostat do stavu projektu v nějakém určitém commitu — víme-li, jaký má identifikátor, můžeme jednoduše použít příkaz `git checkout <identifikator>`. V tomto stavu jsme tzv. detached HEAD, a nemůžeme v něm nic moc dělat. Vznikne nám úplně nelogické oddělení od hlavní linie větví. Můžeme zde začít novou větvi a v ní začít pracovat.

Větve

Větve jsou způsob Gitu na rozdělení jedné práce na projektu do více (kupředu pokračující) verzí. Větve existují v odděleném prostředí a při vytvoření commitu neovlivňujeme další větve.

Interně jsou větve velmi jednoduché — když si představíme, jak vypadá graf commitů, tak větve (angl. branches) jsou takové ukazatele na nějaký commit, ve kterém se nacházejí. Na jednom commitu může být tímto způsobem nalepeno neomezeně větví. Postupem času se stane, že jsou větve velmi daleko v historii od sebe.

V týmových projektech se větve často používají pro různé stavy projektu (produkce, development, beta, ...). Jak se využívají podrobně, si povíme později.

Při vytvoření repozitáře jsme si mohli všimnout, že základní větev se jmenuje master / main. Reálně tato větev nemá žádné speciální funkčnosti a při vytvoření repozitáře ji můžete název změnit.

Vytvoření větve

Lokální větev můžeme vytvořit pomocí příkazu `git branch <jméno>`. Větev můžeme kdykoliv taktéž smazat a to přidáním přepínače `-d`. Pokud nebyla větev do žádné jiné větve spojena (viz. níže), musíme použít přepínač `-D`, který všechny změny zahodí. Vytvořením větve se však do ní nepřesuneme.

Do větve se přesuneme pomocí příkazu `git checkout <branch>`. Přesunutí není možné, pokud máte nějaké soubory rozpracované a nebo dokonce označené jako staged. Po přesunutí budou všechny soubory ze staré větve upraveny / smazány / přidány do podoby, kterou nalezneme v nové větvi.

Při vytvoření commitu v nové větvi se rozběhne historie projektu do více části. Toto rozdělení umí hezky reprezentovat příkaz `git log --graph`.

Všechny větve můžeme zobrazit pomocí příkazů `git branch -v`, které vypíše i poslední comimit, a nebo `git branch` jen se seznamem.

V jaké větvi jsme?

Aby Git věděl, v jaké větvi jsme, používá ukazatel HEAD, který ukazuje na větev (a my víme, že vlastně ukazuje i commit), ve kterém jsme. V příkazu `git log` a `git status` můžeme vidět, v jaké větvi jsme.

Slučování větví

Příkaz `git merge <jméno>` právě sloučení větví způsobí. Příkaz se v AKTUÁLNÍ větvi bude snažit přenést změny série snímků ze zadané větve. Referovaná větev zůstane nepozměněna, aktuální větev bude změněna.

Toto slučování může být jednoduchého rázu (a to tehdy, když změny nejsou komplexní a mají například stejného rodiče) a commity se jen různě posunou. Může dojit taktéž k vytvoření nového commitu (tzv. troj-spojení).

Při slučování mohou nastat také problémy se sloučením. Když ve větvích proběhli velké změny ve stejných souborech a řádcích, Git nebude vědět, co je pravý stav, ve kterém chce po sloučení být. Tento incident se jmenuje **merge conflict**. Merge conflict se musí vyřešit ručně.

Chyba při sloučení

Interně se vytvoří v aktuální větvi všechny změny z větve, ze které spojujeme. Do staged area se vloží již vyřešené sloučení a v modified souborech, ve kterých merge conflicty nastali se vytvoří speciální formule, které vám konflikt popíší. Tuto areu poznáte pomocí šipek a očividného textu. Abyste konflikt vyřešili, musíte text u šípek a rovná se smazat. Po opravení může zůstat první, druhá verze a nebo ruční kombinace dvou zároveň. U šipek se Vám taktéž popisuje, v jaké větvi se změny nacházejí.

```
<<<<<<< HEAD:index.html
<div id="footer">
    <p>Matěj Cajthaml &copy; 1823</p>
</div>
=======
<footer>
    <p>Matěj Cajthaml &copy; 2022</p>
</footer>
>>>>>>> dev:index.html
```

Všechny conflicty naleznete v příkazu `git status`. Až soubory opravíte, můžete se sloučením větví pokračovat pomocí `git commit`.

Vzdálené repozitáře a větve

Protože se Git využívá jako týmový nástroj, často je potřeba, aby existoval vzdálený repozitář, do kterého má každý developer přístup a může v něm dělat změny a stav i stahovat.

Důležité je pochopit, že máme lokální repozitář, který s tím vzdáleným nemusí mít leccos společného. Do lokálního repozitáře si můžeme nalinkovat řadu vzdálených repozitářů, ze kterých můžeme číst a pracovat s nimi. Vzdálený repozitář musí mít jméno, pro hlavní repozitář pro týmový vývoj se často používá `origin`.

Všechny vzdálené repozitáře zobrazíme příkazem `git remote -v`, kde uvidíme každý repozitář dvakrát, jednou pro čtení (fetch) a pro nahrání (push). Vzdálený repozitář přidáme příkazem `git remote add <nazev> <url>`, například tedy `git remote add origin https://github.com/ssps-cajthaml/issues`. Tyto repozitáře lze logicky i smazat pomocí příkazu `git remote rm <nazev>`.

Vzdálené repozitáře obsahují větve, těm říkáme vzdálené větve. Tyto větve lze taktéž mazat. Pokud chceme s větví komunikovat, musíme je stáhnout a implementovat do našich lokálních větví. Na větve odkazujeme dle názvu vzdáleného repozitáře a jejího názvu, tedy `<jméno vz. repozitáře>/<jméno větve>`.

Vzdálené repozitáře můžeme ovlivnit jen různými příkazy a nelze je ovlivňovat jen tak např. z working directory!

Práce se vzdálenými repozitáři

Stáhnutí verze ze vzdáleného repozitáře

Příkaz `git fetch <nazev>` stáhne ze vzdáleného serveru stav repozitáře. Příkaz nic jiného nedělá a neovlivňuje přímo lokální repozitář (ani working directory či cokoliv podobného).

Implementace změn ze vzdáleného repozitáře

Příkaz výše uveden (`git fetch`) změny ze vzdáleného repozitáře pouze stáhne. My je často ale potřebujeme implementovat do našich větví a k tomu slouží příkaz `git pull`. Tento příkaz je taková magie, protože je vlastně složen ze dvou příkazů, a to `git fetch` a `git merge`. Tento příkaz automaticky z větve origin stáhne a pokusí se implementovat změny do aktuální větve. Silně doporučuji používat příkazy odděleně a spojení větví si dělat ručně. Například tedy si stáhnete změny a do aktuální větve implementujete pomocí `git merge vzdalenak/302-cringe`.

Výše uvedená magie s příkazem `git pull` je spíš takový nešvar, který používá věci, které jsme v této lekci (a látce v předmětu) přeskočili. Jednotlivým větvím můžeme totiž nastavovat, jaká je jejich kopie na vzdáleném serveru pomocí tzv. trackingu, více naleznete ZDE.

Nahrání verze do vzdáleného repozitáře

Před odesláním jakékoliv verze musíme změny implementovat či získat pomocí `git fetch` / `git pull`. Pokud větev vzdáleném repozitáři neexistuje, musíme ji vytvořit pomocí přepínače -u s tímto příkazem.

```
git push <vzdálený repozitář> <větev k odeslání>

git push origin master

git push -u origin issue3
```

Klonování repozitáře

Místo založení lokálního repozitáře a komplikovaného přidání vzdáleného repozitáře často naklonujeme pouze vzdálený repozitář a ten automaticky založí ten lokální a ty propojí. Nahráváme rovnou změny, které stahujeme (pull / fetch) a nahráváme (push). Můžeme použít protokol HTTPS nebo SSH (záleží dle Gitové služby).

```
git clone <url repozitáře>

git clone https://github.com/ssps-cajthaml/issues.git
```

Tento příkaz vytvoří složku dle názvu daného repozitáře.

Gitové služby

Zatímco Git je verzovací systém (nástroj), tak existují víceméně známější Gitové služby, které poskytují nad Gitem další kontrolu, ale vnitřně Git využívají a dávají Vám k němu přístup.

GitLab

Jedná se o opensource Gitovou službu. Můžeme si jí nainstalovat na vlastní servery. Je velmi jednoduchá, hezká a má spoustu funkcionalit. Je zdarma v omezeném prostředí.

GitHub

Jedná se o nejznámější Gitovou službu, je však bohužel closedsource. Je zdarma v omezeném prostředí. Pro další části textu budeme uvažovat zejména možnosti, které nabízí ona (avšak např. pro GitLab existují skoro 1:1 alternativy).

Issues

Je to vlastně seznam problémů, nápadů a bugů v daném projektu. Slouží pro lepší orientaci toho, co je potřeba udělat. Když se Issue dokončí, tak se vyřeší a v hlavním seznamu se již nebude zobrazovat. Issue můžeme například v commitech referovat. V Issues lze diskutovat, přiřadit lidi a např. přiřadit značky.

Každá Issue má svoje číslo (čísla se sdílí ještě s pull requesty). Pro Issues se doporučje být konstruktivní a jasně psát své požadavky a problémy. Je důležité např. uvést i jaký operační systém používat a další podrobnosti, které vedou k reprodukci problému.

GitHub dovoluje vytvářet vlastní šablony, které donutí uživatele vyplnit různá pole, které zlepšují přehlednost Issues.

Projects

Jedná se o tzv. Kaban board a nebo tabulky, kdy jsou jednotlivé Issues (a dokonce i textové poznámky) rozdělené do několika kategorií. Používá se pro zjednodušení čtení issues a určení priorit. Github dovoluje tyto boardy automatizován různými akcemi při přesunu atp.

Pull Request

Dovoluje Vám požádat repozitář (resp. jeho majitele) o to, aby z Vašeho repozitáře (resp. větve) stáhli změny a implementovali je do jejich repozitáře.

I když jsou repozitáře veřejné, neznamená to, že do nich (logicky) můžete zapisovat. Proto existuje možnost jak je spojit. Jako v Issues, můžeme do nich psát diskuze, mají různé vlastnosti a přiřazení. Taktéž mají číslo, které můžeme různě referovat. Je administrátoři (resp. uživatelé s právy) mohou pullrequest schválit a kód implementovat.

Pokud chcete do veřejných repozitářů přidávat kód, často při vytvoření prvního pullrequestu budete požádání o podepsání smlouvy (SLA agreement), ve kterém se vzdáte části svých práv na kód. Například souhlasíte s tím, že kód bude vždy veřejný, může být dále upravován, používán pro komerční účely a že tento krok nelze vzít zpět. Každá společnost má tuto smlouvu jinou.

I když je repozitář veřejně, neznamená to, že ke kódu máme jakékoliv práva. Vždy je důležité zkontrolovat licenci, kterou často naleznete v souboru LICENSE.

Releases

Jedná se o způsob vydání vašeho programu či projektu. Automaticky se přikládá kód a ukazují se na hlavní stránce. Při vydání můžeme vložit i vlastní soubory, jako např. zkompilované exe či obdoby. Pro číslování se často používá sémantické verzování, které naleznete v další kapitole.

Milestone

Určení nějaké cíle projektu. Do tohoto cíle, které má často nějakou časovou značku, můžeme určit issues, které se do té doby musí splnit.

Actions

Jedná se o velmi silnou funkcionalitu, která dovoluje buď ručně nebo automaticky spouštět různé skripty a kódy v omezeném prostředí. Nejčastěji se používá např. na otestování kódu, nahrání aplikace na server, nebo třeba pro uzavíraní starých (resp. neaktivních) issues. Pro každé spuštění můžeme vidět jaké práce se konají, logy a podobně. GitHub má speciální nastavovací YAML soubor, ve kterém můžeme využívat již předpřipravené části od komunity.

Správná dokumentace

Sémantické verzování

Jedná se o doporučení, jak správně pojmenovávat (číslovat) verze projektů. Přesné doporučení naleznete ZDE. Zde si ukážeme pouze základy.

Sémantické verzování používá tři čísla, kde se každé z nich v různých chvílích zvětšuje, ve formátu `MAJOR.MINOR.PATCH`. Jednotlivé čísla znamenají:

- `MAJOR` — zvýšení právě tehdy, když nastala změna, která není zpětně kompatibilní
- `MINOR` — zvýšení právě tehdy, když se přidala nová funkcionalita, ale kompatibilita zůstala zachována
- `PATCH` — zvýšení právě tehdy, když se opravila chyba, ale kompatibilita zůstala zachována

Při zvýšení čísel se ostatní čísla vyresetují na nulovou hodnotu. Za těmito čísly se může nacházet ještě předběžné verze a metadata a to ve formátu `-*+*`. Za obě hvězdy lze napsat jakýkoliv text, ale obecně je první např. `alfa`, `beta`, `gamma` a druhá třeba číslo identifikátoru.

Toto doporučení je často různě přeformátováno, kde např. `MAJOR` znamená velké přepsání aplikace, `MINOR` velký posun aplikace, a `PATCH` malý posun či oprava chyb. Nejdůležitější vlastností tohoto verzování je jeho možnost seřazení, mělo by tedy platit, že verze `1.1.1` je zdaleka starší verze, než `12.3.1`.

Dokumentace

Pro projekt je důležité, abyste psali dokumentaci. Dokumentaci lze psát přímo v kódu (která ale slouží spíše pro vývoj aplikace), je však potřeba myslet i na externí dokumentaci, jako jsou návody na spuštění aplikace a podobně. Z kódu lze vytvořit webovou dokumentaci například pomoci Doxygenu a obdob.

Gitové nástroje dovolují vytvářet tzv. Wiki, ve kterých můžete tvořit různé stránky ve formátu Markdown. Gitové nástroje mají taktéž speciální soubor README.*, který se zobrazí při načtení repozitáře (resp. jakékoliv složky). Můžeme zde taktéž používat markdown. V souboru README v kořenovém adresáři repozitáře by měly být ty nejpodstatnější informace projektu. Například jak se projekt používá, k čemu slouží a poskytnout další odkazy na zdroje.

Zásady Gitu

Nyní si představíme nějaké klíčové zásady práce s Gitem, které v praxi určitě potkáte. Pro každé tvrzení bude v této stránce jeden odstavec, kde na začátku je tučně právě dané tvrzení.

Každý commit by měl být funkční (resp. spustitelný). Má-li Git poskytovat možnost verzování, musí být taktéž schopný návratu do commitu. Pokud v historii existuje rozbitý (nefunkční) commit, tak návratem do něj by se projekt mohl rozbít. Proto je důležité, aby commit reprezentovat jednu celistvou změnu a nebyl rozdělený do více commitů.

Zpráva commitu by měla reprezentovat co se v commitu stalo. Historii v Gitu lze jednoduše zobrazovat a v případě kryptických či nelogických zpráv commitů se může stát, že si daný čtenář bude muset celý commit otevřít a podívat se na dané změny. To by mohlo být v nějakých projektech časově náročné.

Je důležité znát firemní (nebo personální) styl práce, syntaxe a sémantiky. Názvy commitů by měli být konzistentní (např. v přítomném čase, anglicky, ...). Budeme-li konzistentní, kdokoliv, kdo projekt zná, se ihned v naší práci vyzná. Znovu budeme časově efektivní.

Historie repozitáře by se měla číst jako kniha a *nezveřejňovat nedokončené části*. Tuto zásadu nám již naše spojování větví rozbilo. V případě komplikovaných sjednocení se vytváří commit, který je vlastně pozůstatkem spojování a při čtení již přestává dávat smysl. To můžeme vyřešit tzv. rebase — přeskládáním, o kterém jsme si nepovídali.

Typy větví

Projekty obvykle používají nějaké typy větví, aby byli co nejvíce konzistentní. Nejčastěji se rozdělují do dvou typů a to stálé (master, develop, next), které zůstávají v projektu napořád a

reprezentují různé hlavní verze projektu, které jsou různě zveřejňované. Druhý typ jsou krátkodobé, které po svém využití umírají. Jedná se např. o issues, experimenty či nápady. Pro issues se často vytvářejí různé speciální názvy větví, například dle jejich identifikátoru a krátkého popisu. Pro issue #23: consistent formatting by se vytvořila např. větev `23-formatting`.

Git klienti

Git klient je software (program) a nebo webová stránka, které zjednodušuje práci s repozitáři. Pro její funkčnost musíme mít Git v CLI. Hlavním důvodem, proč se Git klienti používají je to, že uživatel se nerad dívá do příkazové řádky, kde vidí pouze nějaký text. Díky klientovi můžeme velmi dobře vizualizovat, jak historie Gitu vypadá a klikat jen na tlačítka. Git klienti taky skrývají spoustu věcí, které uživatel nemusí chápat, protože příkazy za něj volají oni. To vede k tomu, že tyto nástroje používají i úplní začátečníci.

Známí Gitoví klienti:

- GitKraken
- GitHub Desktop
- Sourcetree
- **v IDE**

Výhody:

- je vizuálně vidět, v jakém stavu repozitář i projekt je
- lze jednoduše vidět jednotlivé soubory a změny v commitech
- pro základní práci intuitivní
- často jednoduchý způsob řešení merge conflictů
- vše to píše za Vás

Nevýhody:

- píše to všechno za Vás
- často klikáte na něco, čemu vlastně nerozumíte
- není dobré na pochopení (nechápete, co dané tlačítko vlastně přesně dělá)
- často není zdarma, různá podpora

Co jsme s Gitem neprobrali

Za tak málo hodin (a ještě na střední škole) nemůžeme stihnout všechnu látku Gitu. Počítám s tím, že jestli máte zájem o další věci Gitu, že se je naučíte sami buď z dříve referované knížky, nebo obecně z internetu při práci. Níže je seznam věcí, které jsme neprobrali a nebo jednoduše řekli zjednodušeně:

- vnitřnosti Gitu, Git je velmi pokročilý nástroj a vnitřně je důležité znát další věci, jako je například to, jak funguje resetování, reference, protokoly, obnova dat, a spousta dalších
- hooks, což jsou speciální skripty, které se na serveru / počítači spouští při commitu, pushi a nebo při jakékoliv jiné akci. Tímto můžete velmi hezky modifikovat přímo Váš development cycle

- co vlastně .git složka obsahuje za věci, protože to velmi závisí na tom, jak rozumíme vnitřnostem Gitu
- různé další příkazy, jako:
  - stash — uchování aktuálních změn do speciálního balíčku, který můžeme otevřít
  - blame — zjištění, co se s nějakým řádkem v průběhu verze stalo
  - přidávání jednotlivých řádků jako staged
- resetování změn podrobně, sice víme příkazy, ale nevíme jak to přesně funguje, třeba resetování na určitý commit
- různé další způsoby spojování změn — místo sjednocení commitů můžeme použít přeskládání, neboli rebase, či například squash
- submoduly, které nám dovolí vložit do repozitáře jiný repozitář
- a určitě dalších padesát věci, na které jsem jenom zapomněl