



Kontrola Wersji dla Przyszłych Projektów

Miłosz Baczewski, Mateusz Maszczyński

Plan Prezentacji

1

Wprowadzenie do Git

Czym jest Git i dlaczego jest ważny



2

Podstawowe pojęcia

Repozytorium, commit, staging area, gałęzie



3

Podstawowy workflow

Cykl pracy: add, commit, push, pull



4

Mini Demo

Praktyczny przykład operacji Git



5

Praca zespołowa

Repozytoria zdalne, współpraca



6

GitHub

Platforma hostingowa i funkcje



7

Rozwiązywanie konfliktów

Jak radzić sobie z konfliktami



8

Zaawansowane funkcje, Przydatne Komendy

Rebase, stash, cherry-pick



9

Bezpieczeństwo w Git

Dobre praktyki i ochrona danych



10

Git w projektach AI

DVC, GitHub Copilot i narzędzia



11

Najlepsze praktyki

Wskazówki dla efektywnej pracy



12

Zadanie praktyczne

Sprawdź swoją wiedzę



Wprowadzenie - Czym jest Git?

System Kontroli Wersji

Git to rozproszony system kontroli wersji stworzony przez Linusa Torvaldsa w 2005 roku dla rozwoju jądra Linux. System kontroli wersji to narzędzie, które rejestruje zmiany w plikach w czasie, umożliwiając powrót do konkretnych wersji później.

W przeciwieństwie do scentralizowanych systemów kontroli wersji (jak SVN), Git jest rozproszony - każdy programista ma pełną kopię repozytorium na swoim komputerze, co umożliwia pracę offline i zwiększa bezpieczeństwo danych.



Główne zalety Gita:

- ✓ Szybkość i wydajność
- ✓ Rozproszony charakter (praca offline)
- ✓ Integralność danych
- ✓ Wsparcie dla nieliniowego rozwoju (gałęzie)
- ✓ Kompatybilność z istniejącymi protokołami

Podstawowe Pojęcia Gita



Repozytorium (Repo)

Miejsce przechowywania historii projektu. Zawiera wszystkie pliki projektu oraz pełną historię zmian.



Commit

Zapis stanu projektu w określonym momencie. Każdy commit ma unikalny identyfikator, autora i wiadomość.



Gałąź (Branch)

Równoległa linia rozwoju. Pozwala na pracę nad różnymi funkcjami jednocześnie bez wpływania na główną linię.



Scalanie (Merge)

Proces łączenia zmian z różnych gałęzi. Pozwala na integrację pracy wykonanej równolegle.

Trzy Obszary Gita

1

Katalog Roboczy
(Working Directory)



2

Obszar Przygotowania
(Staging Area)



3

Repozytorium
(Repository)

Podstawowy Przepływ Pracy z Gitem

Najważniejsze Komendy

`git init`

Inicjalizuje nowe repozytorium Git w bieżącym katalogu, tworząc ukryty folder `.git` zawierający wszystkie niezbędne pliki.

`git add`

Dodaje zmiany do obszaru przygotowania (staging area). Można dodać konkretny plik (`git add plik.txt`) lub wszystkie zmiany (`git add .`).

`git commit`

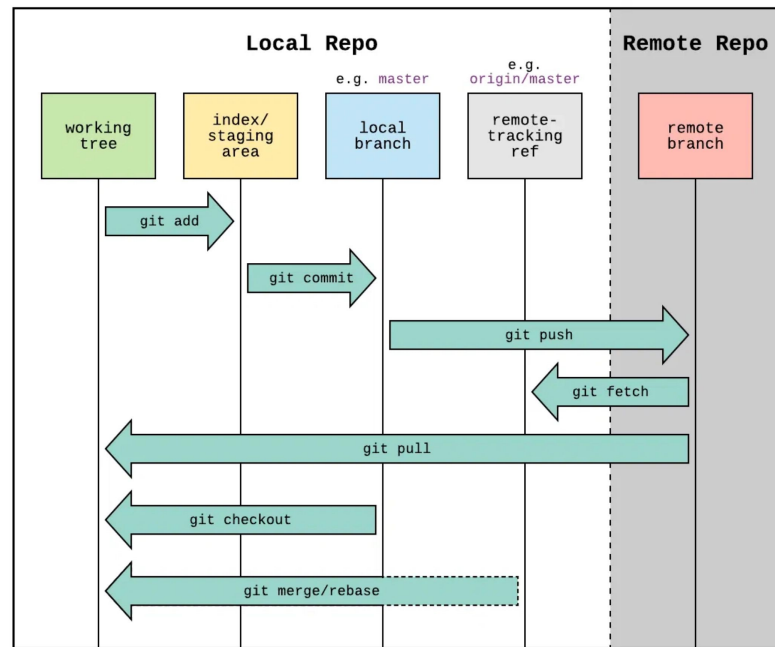
Zapisuje przygotowane zmiany do repozytorium z opisową wiadomością: `git commit -m "Opis zmian"`.

`git status`

Pokazuje stan katalogu roboczego - które pliki zostały zmodyfikowane, które są w obszarze przygotowania, a które nie są śledzone.

`git log`

Wyświetla historię commitów w repozytorium, pokazując autorów, daty i wiadomości commitów.



Mini Demo - Podstawowe Operacje Git

1 Inicjalizacja repozytorium

```
$ mkdir moj-projekt  
$ cd moj-projekt  
$ git init  
Initialized empty Git repository in /home/user/moj-projekt/.git/
```

Tworzymy katalog i inicjalizujemy repozytorium Git.

3 Pierwszy commit

```
$ git commit -m "Pierwszy commit: dodanie pliku README"  
[master (root-commit) f7d2a3c] Pierwszy commit: dodanie pliku README  
1 file changed, 1 insertion(+)
```

Zapisujemy zmiany w repozytorium, tworząc pierwszy commit.

5 Przeglądanie historii commitów

```
$ git log  
commit 8a3b5c7d4e2f1a0b9c8d7e6f5a4b3c2d1e0f9a8b (HEAD -> master)  
Author: Jan Kowalski <jan.kowalski@example.com>  
Dodanie opisu projektu  
  
commit f7d2a3c4b5e6d7f8a9b0c1d2e3f4a5b6c7d8e9f0  
Pierwszy commit: dodanie pliku README
```

Wyświetlamy historię commitów, pokazującą chronologiczną listę zmian w repozytorium.

2 Dodawanie pliku

```
$ echo "# Mój Projekt" > README.md  
$ git status  
Untracked files:  
  README.md  
$ git add README.md
```

Tworzymy plik README.md i dodajemy go do obszaru przygotowania.

4 Modyfikacja i kolejny commit

```
$ echo "To jest projekt demonstracyjny." >> README.md  
$ git add README.md  
$ git commit -m "Dodanie opisu projektu"
```

Modyfikujemy plik i commitujemy tę zmianę.

Praca Zespołowa z Repozytoriami Zdalnymi

Podstawowe Komendy do Pracy Zdalnej

Komenda	Opis
<code>git clone</code>	Pobiera kopię zdalnego repozytorium na lokalny komputer
<code>git push</code>	Wysyła lokalne commity do zdalnego repozytorium
<code>git pull</code>	Pobiera zmiany ze zdalnego repozytorium i scala je z lokalną gałęzią
<code>git fetch</code>	Pobiera zmiany ze zdalnego repozytorium bez scalania
<code>git remote</code>	Zarządza połączeniami ze zdalnymi repozytoriami



Popularne Platformy Hostingowe



Rozwiązywanie Konfliktów

⚠ Czym są konflikty w Gicie?

Konflikty pojawiają się, gdy Git nie może automatycznie połączyć zmian z różnych gałęzi. Najczęściej występują, gdy dwie osoby zmodyfikowały tę samą linię w pliku lub gdy jeden programista usunął plik, a drugi go zmodyfikował.

☰ Proces rozwiązywania konfliktów

- 1 Git sygnalizuje konflikt podczas próby scalenia
- 2 Otwórz pliki z konfliktami i znajdź znaczniki konfliktu
- 3 Edytuj pliki, aby zawierały pożądaną wersję kodu
- 4 Dodaj zmodyfikowane pliki (git add)
- 5 Zakończ proces scalania (git commit)

</> Przykład konfliktu

```
<<<<<< HEAD
function powitanie() {
  console.log("Witaj świecie!");
}
=====
function powitanie() {
  console.log("Cześć wszystkim!");
}
>>>>>> feature-branch
```

Po rozwiązaniu:

```
function powitanie() {
  console.log("Witaj wszystkim!");
}
```

💡 Zapobieganie konfliktom

- ✓ Często pobieraj zmiany z głównej gałęzi
- ✓ Pracuj na małych, izolowanych funkcjonalnościach
- ✓ Komunikuj się z zespołem o modyfikowanych plikach

Zaawansowane Funkcje Gita



Rebase

Alternatywa dla merge, która przepisuje historię commitów. Zamiast tworzyć nowy commit scalający, przenosi lub łączy sekwencję commitów na nową bazę.

```
git rebase master
```



Cherry-pick

Pozwala na wybieranie konkretnych commitów z jednej gałęzi i aplikowanie ich do innej. Przydatne, gdy potrzebujesz tylko określonych zmian.

```
git cherry-pick <commit-hash>
```



Stash

Tymczasowo zapisuje zmiany, które nie są gotowe do commita. Pozwala na przełączenie kontekstu pracy bez commitowania niedokończonych zmian.

```
git stash save "opis zmian"
```



Reset vs Revert

Reset cofa stan repozytorium do wcześniejszego commita, usuwając historię.

```
git reset --hard HEAD~1
```

Revert tworzy nowy commit, który cofa zmiany z poprzedniego commita.

```
git revert HEAD
```



Git Hooks

Skrypty uruchamiane automatycznie w odpowiedzi na określone zdarzenia w Gicie. Mogą być używane do automatyzacji zadań, takich jak sprawdzanie jakości kodu przed commitem, uruchamianie testów przed pushem, czy formatowanie kodu. Znajdują się w katalogu `.git/hooks` w repozytorium.

Przydatne Komendy Git



Poprawianie ostatniego commita

```
git commit --amend -m "Poprawiona wiadomość"
```

Zmienia wiadomość ostatniego commita lub dodaje zapomniane pliki.



Cofanie zmian w plikach

```
git restore plik.txt  
git restore --staged plik.txt
```

Przywraca plik do stanu z ostatniego commita. --staged usuwa ze staging area.



Zarządzanie gałęziami

```
git branch -a  
git branch -d nazwa-galezi  
git branch -m stara nowa
```

-a wszystkie gałęzie, -d usuwa, -m zmienia nazwę.



Tagowanie wersji

```
git tag v1.0.0  
git tag -a v1.0.0 -m "Wersja 1.0"
```

Tworzy tag dla wersji projektu. -a tworzy tag z adnotacją.



Porównywanie zmian

```
git diff  
git diff --staged  
git diff galaz1 galaz2
```

Pokazuje różnice. Bez opcji - niezacomitowane, --staged - w staging area.



Przeszukiwanie historii

```
git log --oneline  
git log --graph --all
```

--oneline zwięzły widok, --graph graficzna reprezentacja gałęzi.



Tymczasowe przechowanie zmian

```
git stash  
git stash pop  
git stash list
```

Zapisuje zmiany na stosie. pop przywraca ostatnie zmiany.



Cofanie commitów

```
git reset --soft HEAD~1  
git reset --hard HEAD~1  
git revert commit-hash
```

--soft cofa commit, zachowując zmiany. --hard usuwa zmiany.

GitHub - Platforma dla Programistów



Największa platforma hostingowa dla repozytoriów Git, należąca do Microsoft od 2018 roku

Statystyki GitHub

100M+

Użytkowników

420M+

Repozytoriów


90M+


Projektów

330M+

Pull Requestów rocznie

Kluczowe funkcje

 **Pull Requests** - mechanizm do przeglądu i dyskusji nad zmianami w kodzie

 **Issues** - system śledzenia błędów i zarządzania zadaniami

 **GitHub Actions** - automatyzacja CI/CD i przepływów pracy

 **GitHub Pages** - darmowy hosting dla stron statycznych

 **Security** - skanowanie bezpieczeństwa i zarządzanie zależnościami





 **Collaboration** - narzędzia do pracy zespołowej i code review

Dla studentów

GitHub oferuje **GitHub Student Developer Pack** - darmowy dostęp do narzędzi premium dla studentów, w tym GitHub Pro, GitHub Copilot i wiele innych!

Bezpieczeństwo w Git

Najczęstsze zagrożenia

-  **Dane wrażliwe w historii** - hasła, klucze API, tokeny dostępu, które raz commitowane pozostają w historii repozytorium
-  **Nieautoryzowany dostęp** - brak odpowiednich uprawnień do repozytoriów lub brak uwierzytelniania dwuskładnikowego
-  **Złośliwy kod** - wprowadzanie szkodliwego kodu przez nieautoryzowane osoby lub poprzez zależności
-  **Ataki typu supply chain** - kompromitacja zależności lub narzędzi używanych w projekcie



Pamiętaj!

Usunięcie pliku z najnowszego commita **nie usuwa go z historii**. Dane wrażliwe mogą być nadal dostępne w poprzednich commitach.

Wskazówka

Dla projektów AI/ML, rozważ użycie narzędzi takich jak DVC (Data Version Control) do wersjonowania danych i modeli, oraz MLflow do śledzenia eksperymentów. Integrują się one z Gitem, tworząc kompletny system kontroli wersji dla projektów uczenia maszynowego.

Dobre praktyki bezpieczeństwa

-  **Używaj .gitignore** - ignoruj pliki konfiguracyjne zawierające dane wrażliwe
-  **Stosuj zmienne środowiskowe** - przechowuj dane wrażliwe w zmiennych środowiskowych zamiast w kodzie
-  **Używaj menedżerów sekretów** - narzędzia takie jak HashiCorp Vault, AWS Secrets Manager
-  **Włącz uwierzytelnianie dwuskładnikowe** - dodatkowa warstwa zabezpieczeń dla kont GitHub/GitLab

```
# Przykładowy plik .gitignore dla bezpieczeństwa
# Pliki konfiguracyjne z danymi wrażliwymi
config.json
.env
credentials.yml

# Klucze i certyfikaty
*.pem
*.key
*.cert

# Logi i pliki tymczasowe
*.log
tmp/
```

Git a Sztuczna Inteligencja

Wyzwania w Projektach AI



Duże Pliki

Zbiory danych i wytrenowane modele często przekraczają limity Gita. Rozwiązanie: Git LFS (Large File Storage) do przechowywania dużych plików binarnych.



Wersjonowanie Modeli

Śledzenie różnych iteracji modeli ML wymaga specjalnych narzędzi. Git sam w sobie nie śledzi parametrów eksperymentów ani metryk modeli.



Śledzenie Eksperymentów

Potrzeba łączenia kodu, danych, parametrów i wyników w spójny sposób. Standardowy Git nie oferuje narzędzi do śledzenia eksperymentów ML.

Narzędzia AI Wspomagające Git

GitHub Copilot



AI asystent programowania, który sugeruje kod na podstawie kontekstu i komentarzy. Wspiera pracę z repozytoriami Git, pomagając w szybszym tworzeniu kodu.



Generatory Komunikatów Commitów

Narzędzia AI, które automatycznie generują opisowe komunikaty commitów na podstawie wprowadzonych zmian w kodzie (np. OpenCommit, GitKraken AI).

Data Version Control (DVC)



Rozszerzenie dla Gita specjalnie zaprojektowane do projektów ML. Pozwala na wersjonowanie danych, modeli i eksperymentów. Działa na zasadzie "Git dla danych" - śledzi zmiany w dużych plikach bez przechowywania ich w repozytorium Git.

Najlepsze Praktyki w Git

Dobre praktyki commitowania

- ✓ **Commituj często** - małe, logiczne zmiany zamiast dużych, złożonych commitów
- ✓ **Pisz jasne wiadomości** - używaj formy rozkazującej, np. "Dodaj funkcję logowania" zamiast "Dodana funkcja logowania"
- ✓ **Używaj konwencji** - np. "feat: dodaj logowanie", "fix: napraw błąd walidacji"

Współpraca zespołowa

- ✓ **Używaj pull requestów** - umożliwiają code review przed scaleniem zmian
- ✓ **Dokumentuj swoje zmiany** - aktualizuj README i dokumentację projektu
- ✓ **Komunikuj się z zespołem** - informuj o dużych zmianach, które mogą wpłynąć na pracę innych
- ✓ **Używaj issues/zadań** - śledź postęp pracy i przypisuj zadania

Strategie gałęzi

- ✓ **Używaj gałęzi dla nowych funkcji** - twórz osobną gałąź dla każdej nowej funkcjonalności
- ✓ **Utrzymuj czystą główną gałąź** - `main` lub `master` powinna zawsze zawierać stabilny kod
- ✓ **Regularnie aktualizuj swoje gałęzie** - synchronizuj z główną gałęzią, aby uniknąć konfliktów
- ✓ **Usuń nieużywane gałęzie** - po scaleniu zmian, usuń niepotrzebne gałęzie dla zachowania porządku

Narzędzia i automatyzacja

- ✓ **Używaj narzędzi graficznych** - GitKraken, SourceTree, GitHub Desktop dla lepszej wizualizacji
- ✓ **Konfiguruj aliasy** - skróty dla często używanych komend Git
- ✓ **Używaj Git hooks** - automatyzuj testy, linting i formatowanie kodu
- ✓ **Integruj z CI/CD** - automatyczne testy i wdrożenia po commitach

Zadanie Praktyczne

📋 Zadania do wykonania

- 1 Utwórz katalog `moj-projekt` i zainicjalizuj repozytorium Git
- 2 Stwórz plik `README.md` i `.gitignore`
- 3 Dodaj pliki i wykonaj pierwszy commit
- 4 Utwórz gałąź `feature-model` i przełącz się na nią
- 5 Dodaj plik `model.py` z kodem Python, commituj
- 6 Wróć do `main` i scal zmiany z `feature-model`
- 7 Wyświetl historię commitów: `git log`

★ Zadanie bonusowe

Utwórz konto na GitHub i wypchnij projekt na zdalne repozytorium

💡 Wskazówki i podpowiedzi

- ✓ Używaj `git status` po każdym kroku
- ✓ W `.gitignore` dodaj: `*.pyc`, `__pycache__`, `.env`
- ✓ Utworzenie gałęzi: `git checkout -b nazwa-galezi`
- ✓ Scalenie: `git merge nazwa-galezi`
- ✓ Pisz jasne wiadomości commitów w formie rozkazującej
- ✓ Przykład commita: `git commit -m "Dodaj model AI"`

🏆 Cel zadania

Po ukończeniu będziesz potrafił/a:

- Inicjalizować repozytoria Git
- Tworzyć i zarządzać gałęziami
- Wykonywać commity i scalać zmiany
- Pracować z GitHub (bonusowo)

❗ Pamiętaj!

Przed pierwszym użyciem Git wykonaj komendy:

```
git config --global user.name "Twoje Imie"
git config --global user.email "Twoj email"
```

Bibliografia i Źródła



Dokumentacja i książki

- 1 **Pro Git** - Scott Chacon, Ben Straub. Oficjalna książka o Git, dostępna za darmo na git-scm.com/book
- 2 **Git Documentation** - Oficjalna dokumentacja Git. git-scm.com/doc
- 3 **GitHub Docs** - Kompleksowa dokumentacja GitHub. docs.github.com
- 4 **Atlassian Git Tutorials** - Szczegółowe tutoriale i przewodniki. atlassian.com/git/tutorials
- 5 **GitLab Documentation** - Dokumentacja GitLab i CI/CD. docs.gitlab.com

Przydatne narzędzia

GitHub Desktop
desktop.github.com

GitKraken
gitkraken.com



Kursy i zasoby edukacyjne

- 6 **GitHub Learning Lab** - Interaktywne kursy Git i GitHub. lab.github.com
- 7 **Learn Git Branching** - Wizualna nauka Git przez ćwiczenia. learngitbranching.js.org
- 8 **Git Immersion** - Praktyczny przewodnik krok po kroku. gitimmersion.com
- 9 **DVC Documentation** - Data Version Control dla projektów ML. dvc.org/doc
- 10 **GitHub Copilot** - Asystent AI dla programistów. github.com/features/copilot
- 11 **Git Cheat Sheet** - Ściągawka z najważniejszymi komendami. education.github.com/git-cheat-sheet
- 12 **Oh My Git!** - Gra edukacyjna do nauki Git. ohmygit.org