**2 Git - Podstawy pracy**

**Wstęp**

W szkoleniu tym omówimy pierwsze podstawowe polecenia, które są niezbędne do pracy z GIT-owym systemem kontroli wersji. Po tej lekcji powinieneś umieć tworzyć i konfigurować repozytoria, zarządzać śledzeniem plików, umieszczać zmiany w poczekalni (stage) oraz je zatwierdzać. Pokażemy ci także, jak pobierać dowolne repozytorium i aktualizować już istniejące. Ponadto nauczymy się konfigurować GITa tak, aby ignorował pewne pliki oraz całe ich grupy według zadanego wzorca. Dzięki temu wszystkiemu będziesz w stanie zarządzać swoimi pierwszymi projektami na GitHubie.

**Publikowanie pierwszego repozytorium GITa**

Projekt GIT-a możemy rozpocząć w dwojaki sposób. Możliwe jest rozpoczęcie go od zera lub sklonowanie już istniejącego ze strony GitHub. Na sam początek zajmijmy się pierwszym przypadkiem:

**git init**

Polecenie to umożliwia utworzenie repozytorium GITa z dowolnego przechowywanego lokalnie projektu. A to dlatego, że git init umożliwia kojarzenie systemowi, że folder, w którym wykonano tę komendę jest GIT-owym repozytorium. Gdybyś się zastanawiał, jak to jest realizowane, to omawiany system tworzy wewnątrz katalogu folder .git. Wspomniałem już o nim przy omawianiu trzech stanów plików w systemie kontroli wersji. Przypomnę Ci tylko, że .git trzyma cały szkielet repozytorium Gita. To w nim znajdują się informacje o ostatnich commitach, przechowywana jest lokalna migawka projektu.

**git status i git add**

Po tym, jak lokalnie zainicjalizowałeś swoje repo, możesz już przemieszczać znajdujące się w nim zasoby pomiędzy różnymi stanami. Pamiętaj jednak, że w takim katalogu muszą się znajdować pewne pliki - folder nie może być pusty. Tak więc, jeżeli inicjalizujesz puste repozytorium, to pamiętaj o dodaniu do niego pierwszych zasobów. Po spełnieniu tego warunku, możesz sprawdzić stan znajdujących się wewnątrz folderu plików - użyj polecenia **git status**.

Git status wyświetla listę plików i katalogów, które znajdują się w stanie working directory lub staged. Polecenie to umożliwia rozróżnienie, w którym stanie znajduje się konkretny zasób na podstawie koloru wyświetlanych nazw. **Kolorem czerwonym** zaznaczone są pliki nieśledzone (czyli te, które znajdują się w pierwszym stanie working directory). **Kolorem zielonym** oznaczone są natomiast zasoby umieszczone już w przechowywalni (stanie stage).

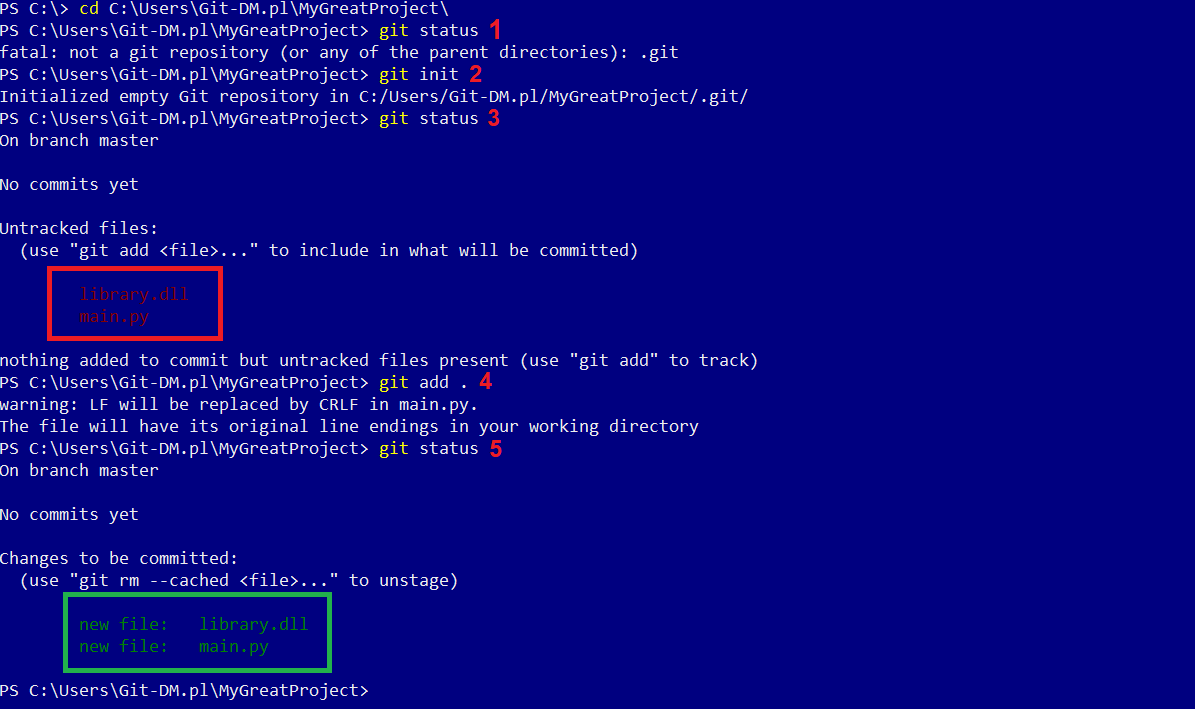
Zasoby ze stanu pierwszego do drugiego możemy przenosić, wykorzystując polecenie **git add**. Chcąc, np. przenieść plik test.txt do przechowywalni, zapiszemy **git add test.txt,** kilka plików o nazwach test2 i test3: **git add test2.txt test3.txt** lub wszystkie zasoby znajdujące się w lokalnym repozytorium: **git add .**

**Przykład:**

Załóżmy, że do utworzonego we wcześniejszym szkoleniu repozytorium na GH - Example1, chcemy wypushować (uploadować) pierwsze zasoby. Zasoby te znajdują się w katalogu pod ścieżką C:\Users\Git-DM.pl\MyGreatProject (ja akurat korzystam z Windowsa, ale na systemie Unixowym, Twój katalog mógłby równie dobrze znajdować się tutaj: ~/Git-DM.pl/MyGreatProject).

W takim katalogu umieściłem już kilka plików projektowych: main.py oraz library.dll. Wypróbujmy na nich trzy nowo poznane polecenia: git init, git status oraz git add.

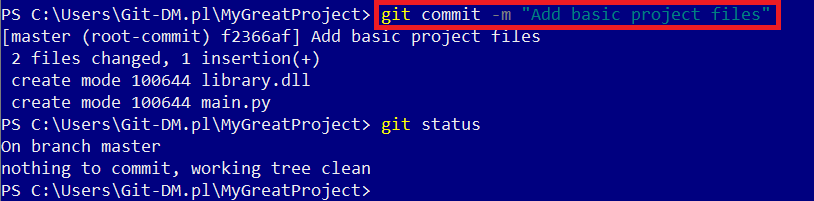
W pierwszej kolejności wykorzystamy polecenie git init, które posłuży nam do inicjalizacji repozytorium GIT-owego (zostanie utworzony ukryty katalog .git). Następnie przeniesiemy wszystkie pliki do stanu stage poleceniem git add. Na bieżąco będziemy również sprawdzali status plików, wpisując polecenie git status.



1. Użyliśmy tutaj polecenia git status. Jak już wiesz służy ono do sprawdzenia statusu plików. Zauważ, jednak, że po wykonaniu tej komendy, otrzymaliśmy komunikat “*fatal: not a git directory (or any of the parent directories): .git*”. To dlatego, że katalog, w którym się znajdujemy - MyGreatProject - nie jest jeszcze traktowany jako repozytorium gita.
2. Właściwa już inicjalizacja lokalnego repozytorium, na którym będziemy mogli wykonywać polecenia GITa.
3. Ponowne sprawdzenie statusu zasobów, zauważ że tym razem nazwy plików wyświetliły się kolorem czerwonym (nie są one śledzone w stanie working directory).
4. Aby to się zadziało, musimy użyć polecenia git add . (przeniesienie wszystkich zasobów do staged).
5. Po kolejnym sprawdzeniu statusu plików, widzimy, że wszystkie zostały przeniesione do przechowywalni (o czym świadczy kolor zielony)

**Git commit**

Następnym koniecznym do poznania poleceniem jest git commit. Komenda ta przenosi pliki i katalogi z przechowywalni do repository. Dodatkowo z każdym commitem związana jest wiadomość (komentarz), którą musimy określić. Spójrz poniżej (przeniesienie plików main.py oraz library.dll określimy komentarzem *“Add basic project files”*):



Zauważ, że komentarz ten dodajemy po fladze -m(skrót od -message). I choć wydaje się to błahostką, to musisz przestrzegać podstawowych zasad związanych z podawaniem wiadomości dla każdego commita:

* Commit message musi mówić, czego konkretnie dotyczy zmiana
* W commitach trzymaj się jednej nomenklatury, np. decyduj się na jeden język - pisz tylko po angielsku lub po polsku, stosuj wielkie i małe litery itp.
* W razie potrzeby, dziel commit message na sekcję subject (krótko opisującą sens zmiany) oraz body (Będący rozwiniętym opisem commita)
* Subject nie powinien przekraczać długości 50 znaków

Po wykonaniu polecenia git status, widzimy napis “nothing to commit, working tree clean” - tak więc nasze pliki znajdują się już w stanie repository. Teraz nie zostało nic innego jak wypushować zmiany do zdalnego repozytorium na GH.

**Git push**

Po commit-cie, nadeszła pora na wypushowanie zmian. Git push służy do uploadowania zmian z lokalnego repozytorium na GH. Do polecenia push musimy również dostarczy dwa parametry, tzw. adres URL projektu, do którego chcemy wypushować zmiany oraz branch\_name, czyli gałąź projektu, której zawartość ma być udostępniona.   
  
Temat branch-ów poruszymy w dalszych szkoleniach. Na ten moment bądź świadomy tego, iż GIT umożliwia programiście pracę na różnych gałęziach (ang. branch) projektu. Domyślnie pracujemy na gałęzi master - nazywanej gałęzią główną i to właśnie jej nazwę będziemy podawali przy pushowaniu zmian.

**Ciekawostka:**

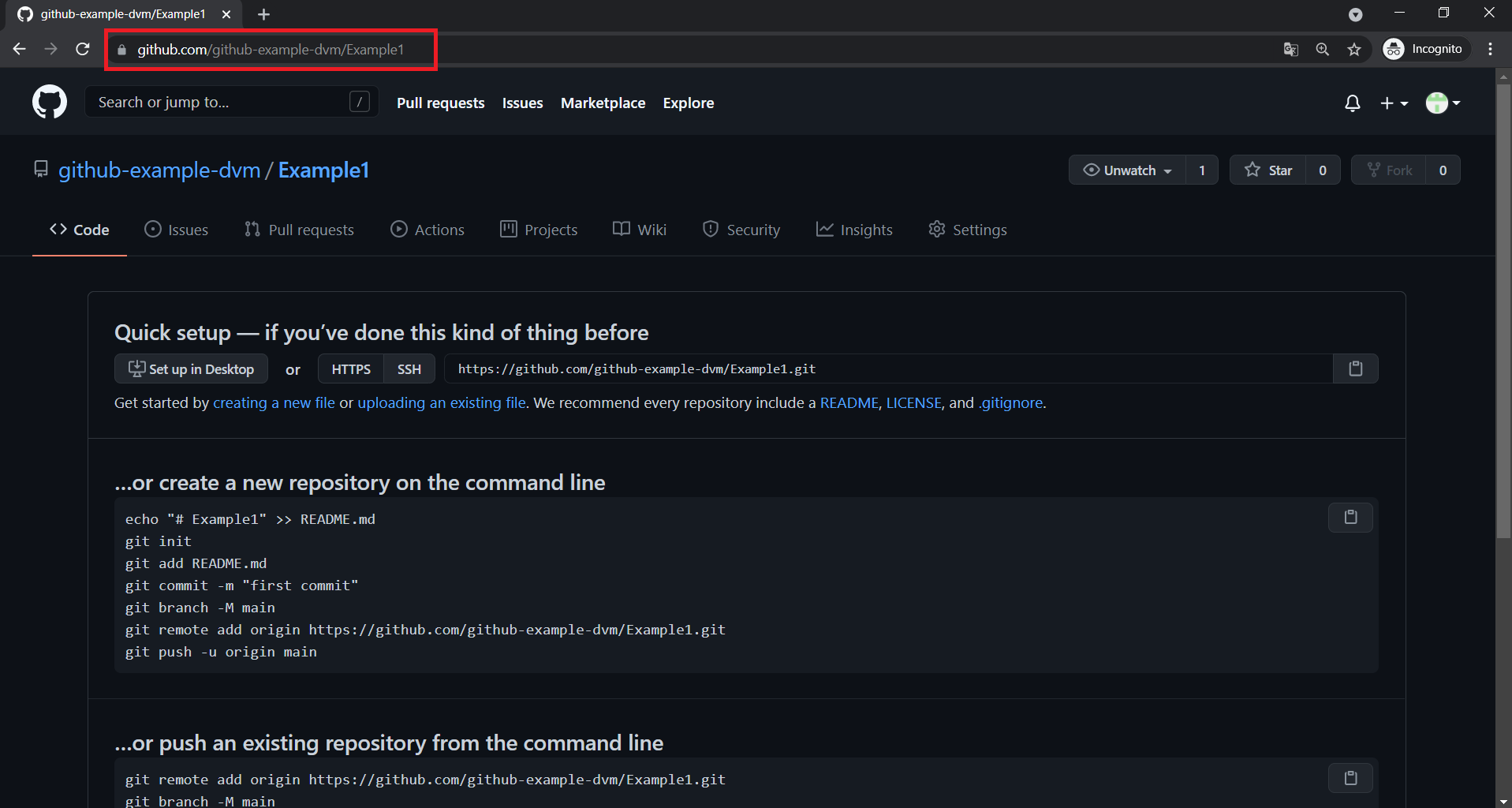
W nowszych wersjach GIT-a gałąź główna może funkcjonować pod nazwą main. A to dlatego, że w październiku 2020 roku, po licznych protestach przeciwko nierówności i dyskryminowaniu czarnoskórych obywateli, zdecydowano się zamienić słowo master na właśnie main.

Jeżeli chcesz się przekonać, na jakiej gałęzi Ty pracujesz, w terminalu wpisz polecenie git branch:

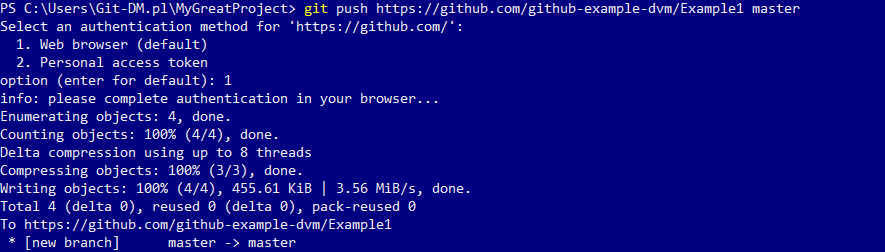


Sprawdźmy, jak będzie wyglądał proces udostępniania zmian do publicznego repozytorium Example1:

1. W pierwszej kolejności należy pobrać adres URL repozytorium na GH. W moim przypadku ma on postać: <https://github.com/github-example-dvm/Example1> (adres repozytorium utworzonego we wcześniejszym szkoleniu).



1. Po pobraniu adresu, możemy śmiało przejść do terminala, udać się do katalogu projektowego i wykonać polecenie: **git push <adres\_URL> master (lub main; w zależności od tego, co wyświetliło polecenie git branch)**

****

Po wykonaniu polecenia, zostaniesz poproszony o potwierdzenie swojego dostępu do konta - będziesz mógł dokonać autoryzacji przez zalogowanie się w przeglądarce do konta lub podanie access token-u. My skorzystaliśmy z pierwszej (domyślnej) opcji. Po zalogowaniu się i potwierdzeniu tożsamości, zmiany zostały wypushowane do zdalnego repozytorium.

| **Uwaga!**  Wraz z końcem sierpnia roku 2021, polityka GIT wprowadziła konieczność logowania się do konta tylko i wyłącznie przy użyciu Access Tokena. W takiej sytuacji za hasło wklej po prostu wartość tak wygenerowanego tokenu, który znajdziesz pod zakładką: <https://github.com/settings/tokens> |
| --- |

**Wskazówka:**

Aby nieco usprawnić swoją pracę, możesz dodatkowo wykonać polecenie, które umożliwi zapamiętanie adresu URL danego repozytorium pod przyjazną nazwą. Dzięki temu nie będziesz musiał za każdym razem pobierać odpowiedniego adresu repozytorium w celu wypushowania zmian. Wystarczy, że w konsoli wpiszesz:

git remote add <dowolna\_nazwa> <adres\_URL>

Za dowolna\_nazwa najczęściej wstawiamy słowo “origin” (ang. pochodzenie). Oczywiście nic nie stoi na przeszkodzie, aby to była dowolna nazwa, jednak w pracy z GIT-em słowo to zostało już głęboko ukorzenione.

Dzięki temu, git push z takiej formy:

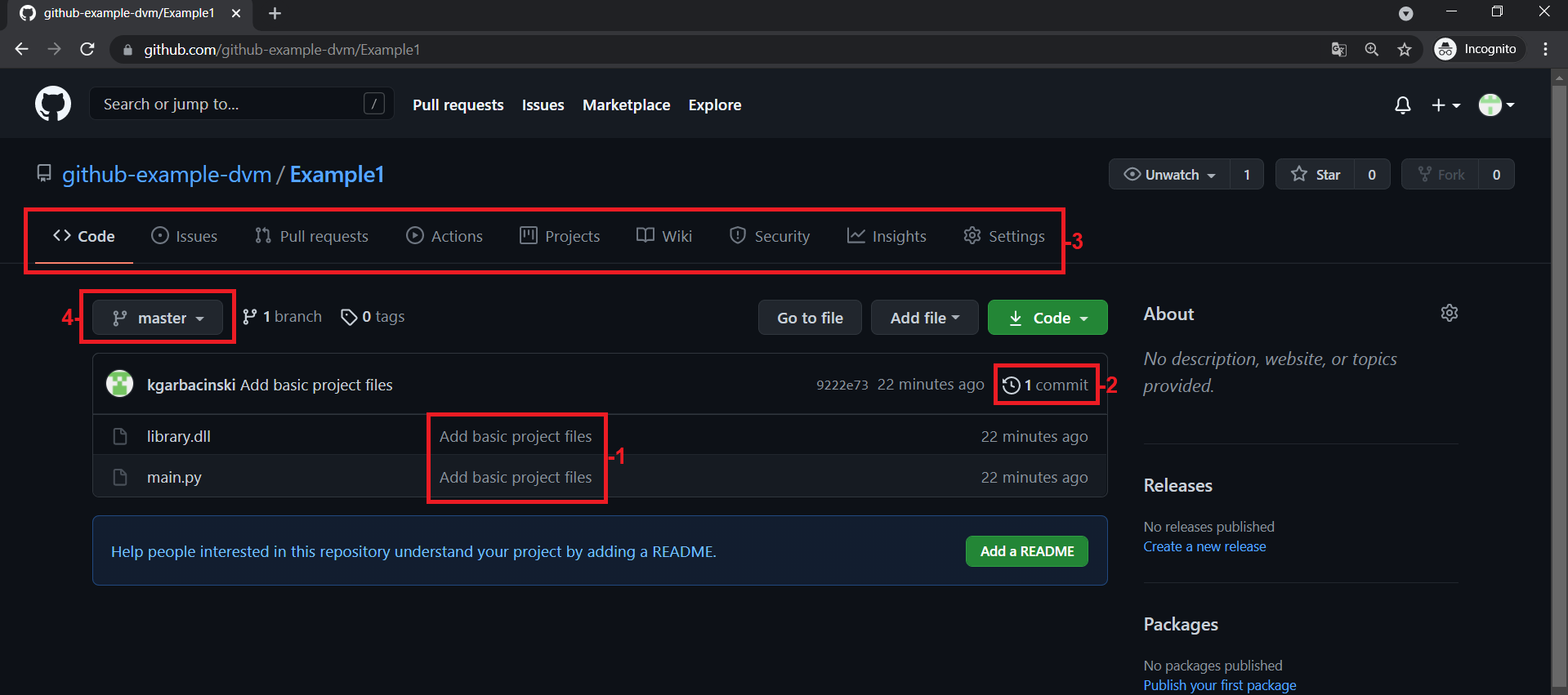
git push <https://github.com/github-example-dvm/Example1> master

Będzie mógł zostać przekształcony do:

git push origin master

**Potwierdzenie zmian**

Teraz, gdy przejdziemy do repozytorium Example1 na GH, ujrzymy dodany nowo dodany projekt. Gratulacje! Udało Ci się wypushować pierwszą zmianę. Daj sobie chwilę na zapoznanie się z interfejsem.



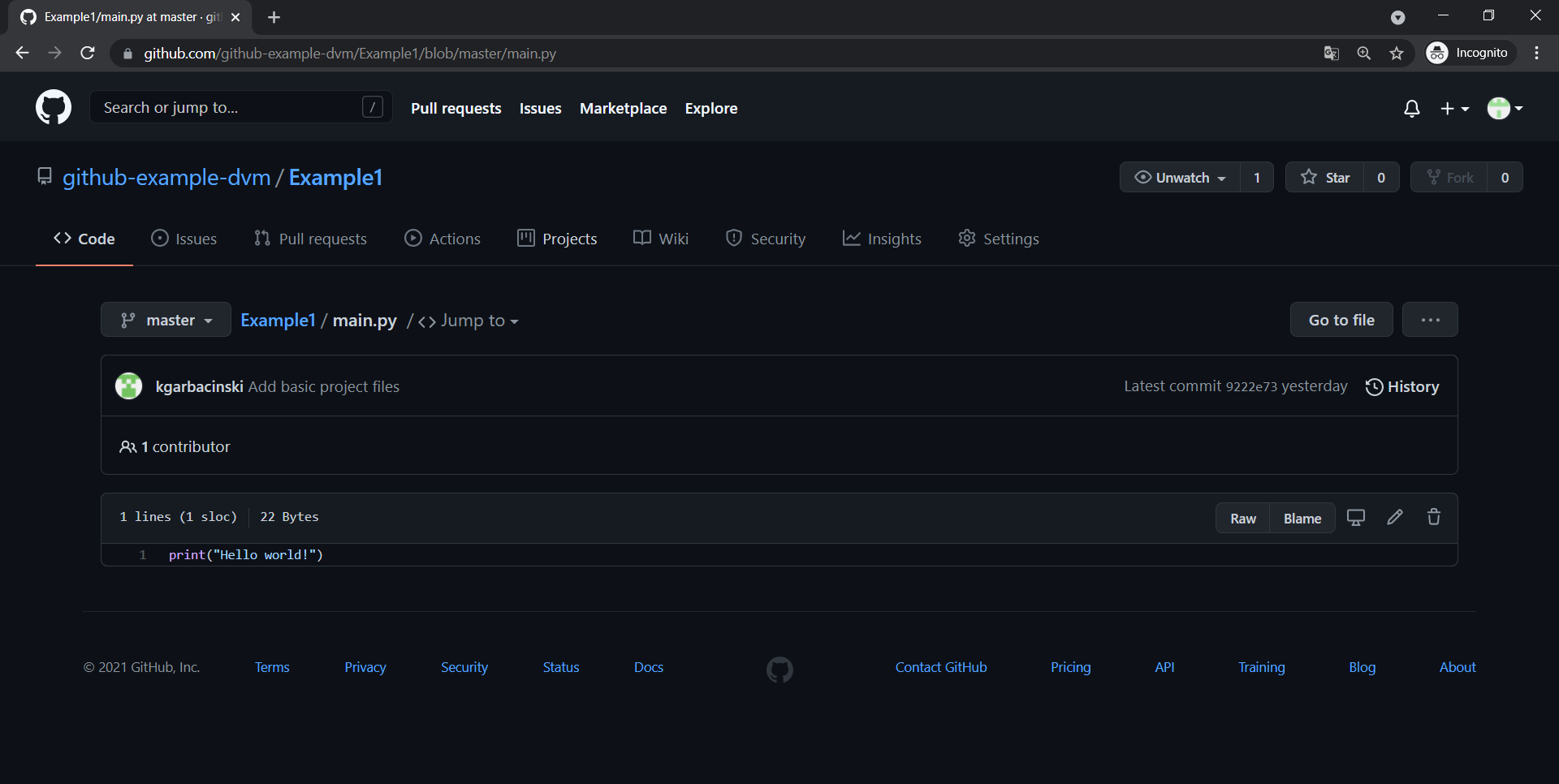
1 - **Commit message**, który dodaliśmy przy przenoszeniu plików ze stanu stage do repository.

2 - **Historia commitów**. Po kliknięciu w ten odnośnik, możesz ujrzeć historię zmian, którą przeprowadziłeś i wypushowałeś w ramach swojego projektu. To właśnie wyjaśnia, dlaczego tak ważne jest przywiązywanie dużej uwagi do poprawności i czytelności tworzonych wiadomości commitów.

3 - **Różne zakładki powiązane z repozytorium**. Powiem szczerze, że na ten moment nie widzę sensu w omawianiu każdej z osobna - przebrniemy przez nie wraz z rozwojem naszych umiejętności

4 - **Zakładka umożliwiająca przechodzenie między różnymi gałęziami projektu**. Jest ona niezwykle przydatna w momencie, gdy pracujemy na kilku gałęziach projektowych. Na ten moment za dużo ona nie wnosi - a to dlatego, że pracujemy tylko na jednej gałęzi master.

Zwróć również uwagę na to, jak proste i intuicyjne jest przeglądanie udostępnionego kodu. Zauważ, jak po kliknięciu w dowolny plik z kodem, możesz łatwo nadzorować jego zawartość.

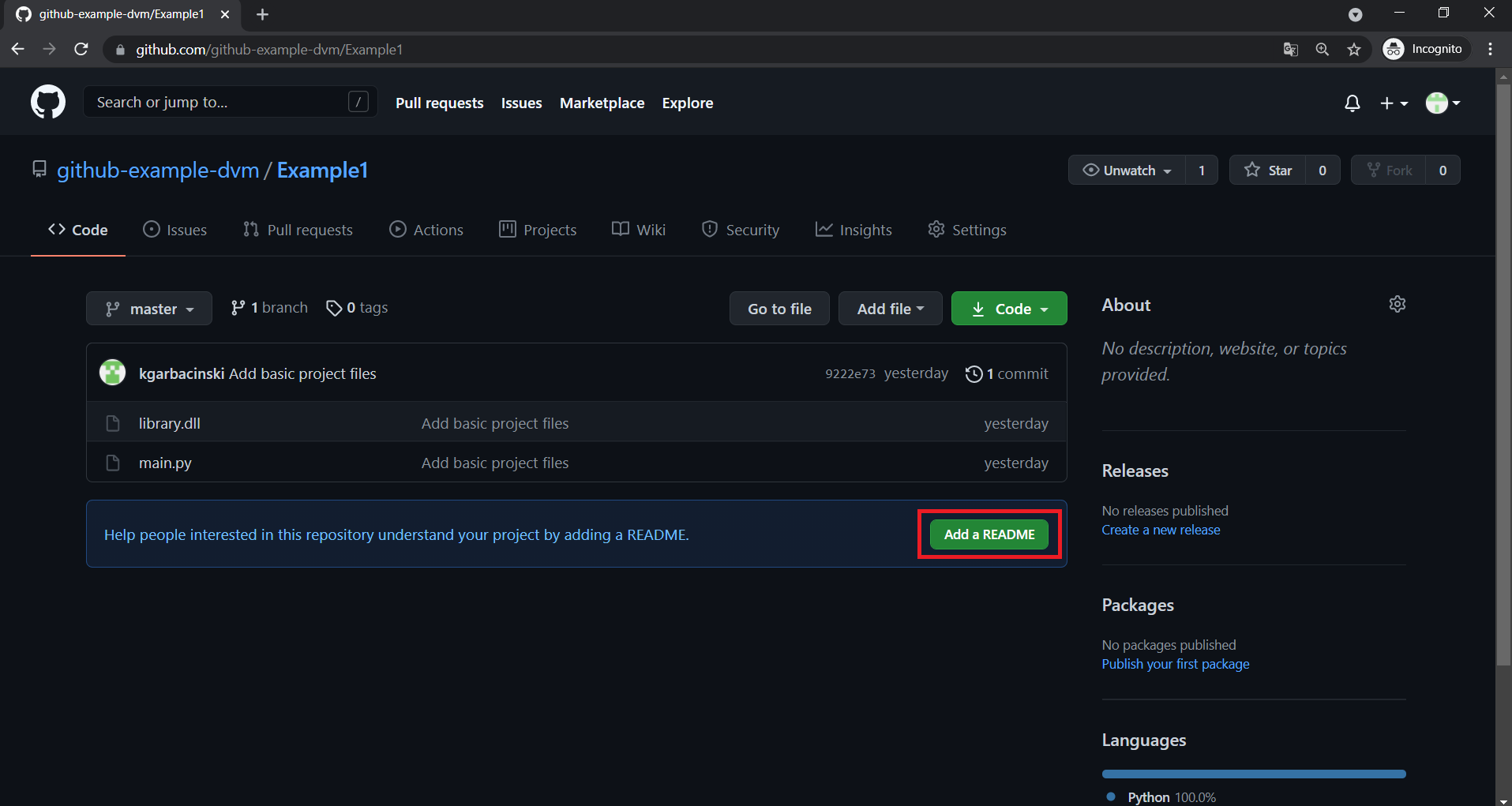


*Zawartość pliku main.py*

**Wprowadzanie zmian z poziomu GH**

Wiesz już, jak publikować zmiany wewnątrz **lokalnego** repozytorium. W tej sekcji chciałbym poruszyć zagadnienie modyfikowania plików z poziomu serwisu GH.

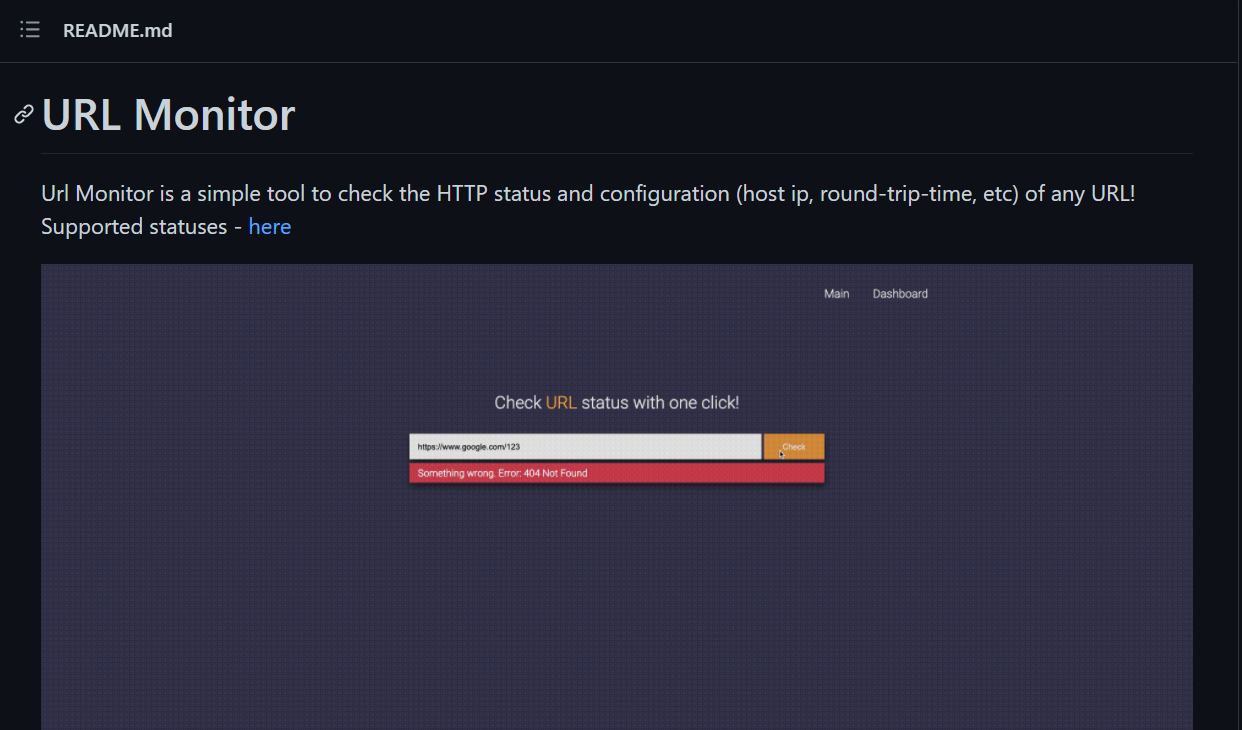
Na początek muszę Cię jednak uświadomić. Edycja plików z poziomu strony GH, nie jest czynnością zalecaną, dlatego raczej wystrzegaj się edycji kodu z poziomu GitHuba. Jedyną sensownie uzasadnioną modyfikacją do przeprowadzenia to dodanie pliku README.md do repozytorium z poziomu interfejsu strony. Możesz tę opcję dostrzec, przechodząc do widoku głównego repozytorium Example1:



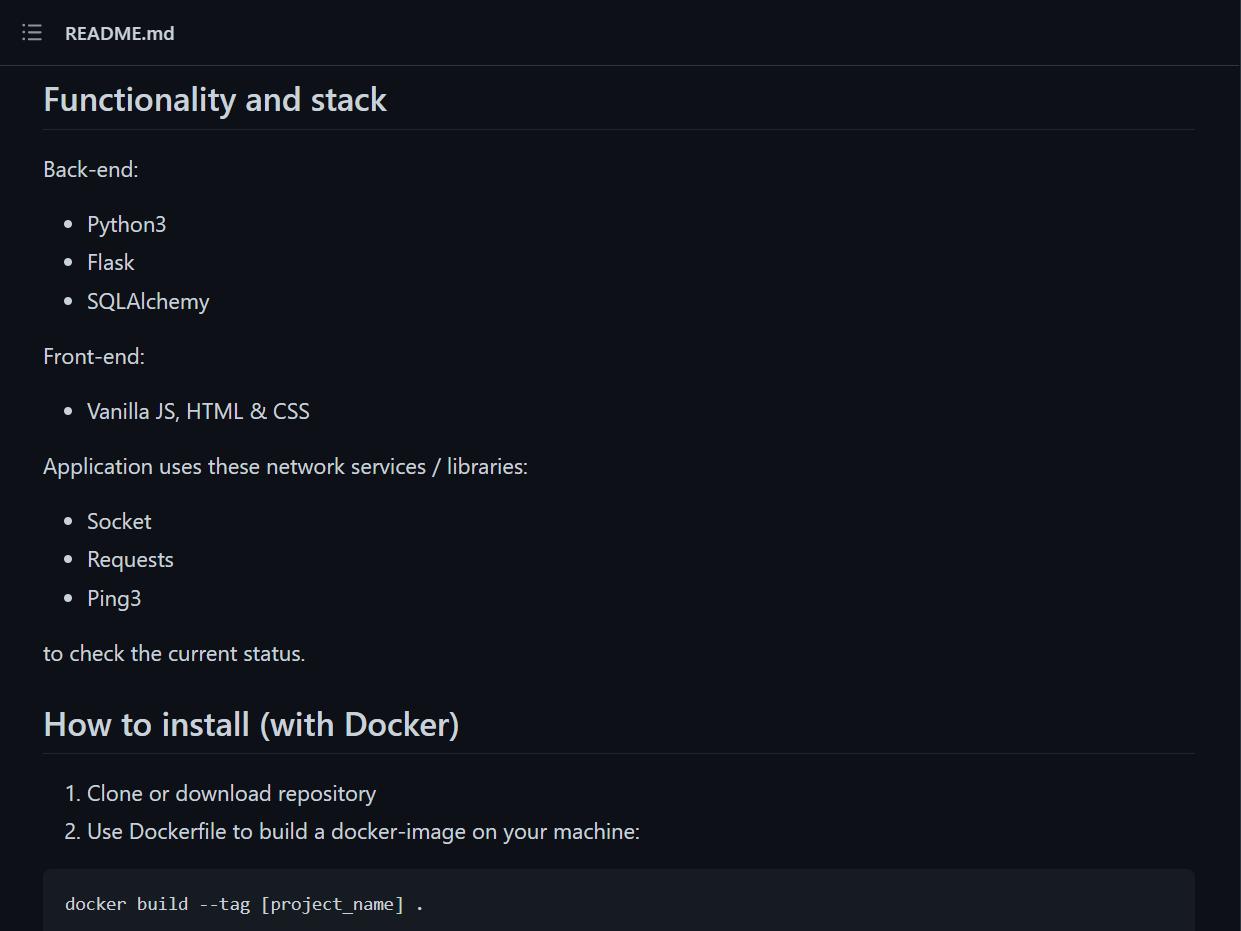
**Tworzenie README**

Tak jak wspomniałem już wcześniej, plik README powinien przechowywać najważniejsze informacje o utworzonym projekcie - m.in jakie biblioteki zostały wykorzystane, jak skonfigurować aplikację itd.

Przykładowy plik README z jednego z projektów naszych uczniów wygląda następująco:



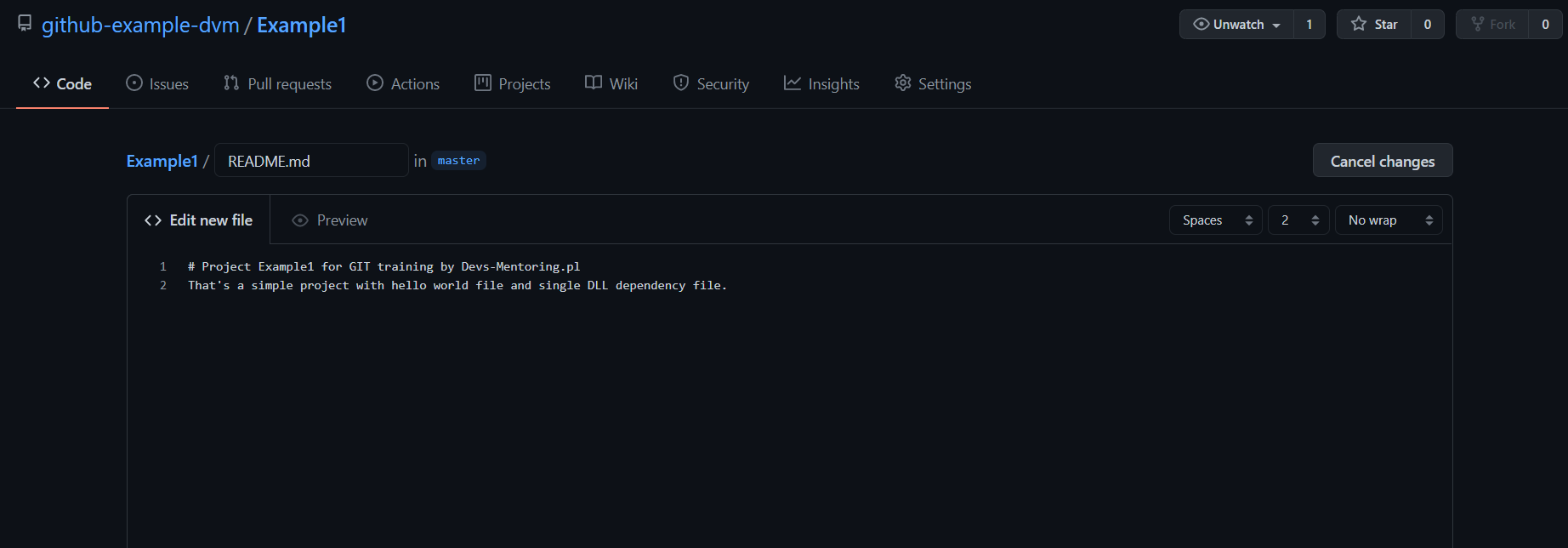
*Plik Readme.md (1)*



*Plik Readme.md (2)*

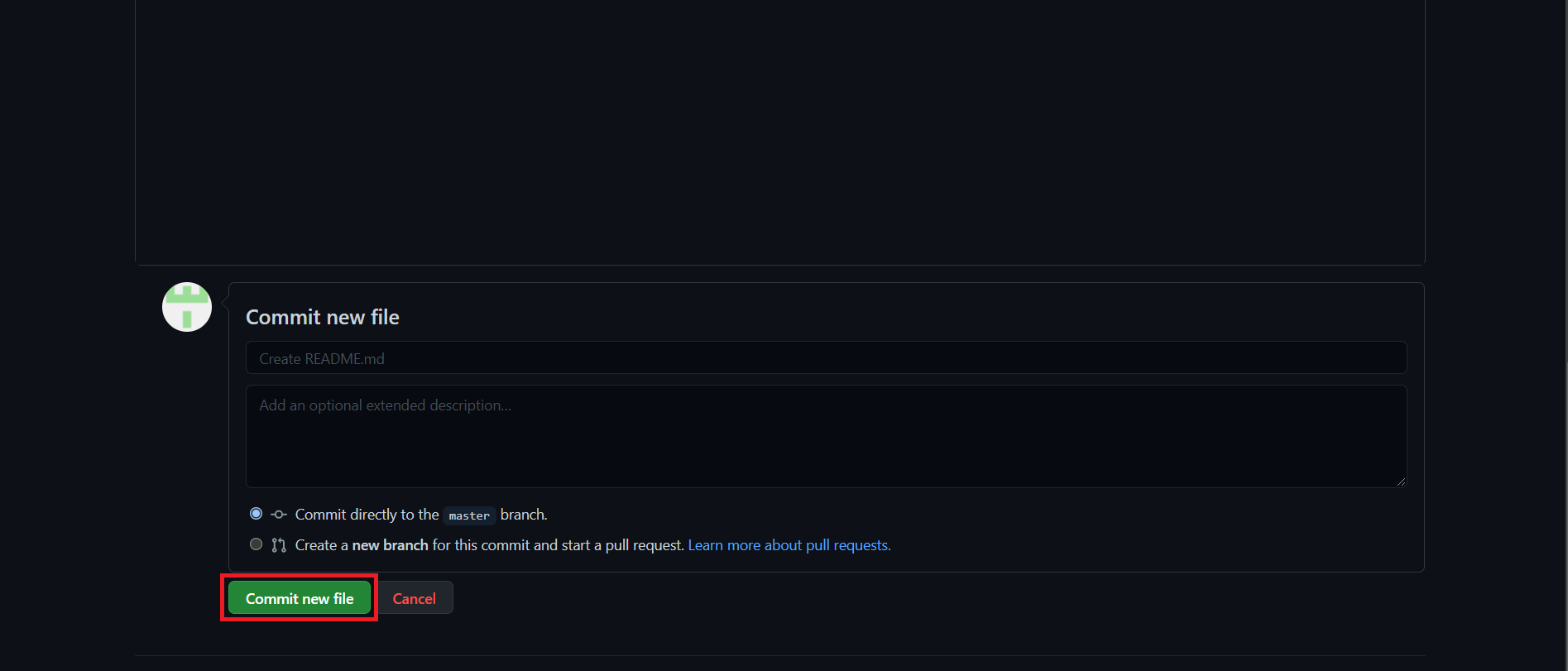
Dodajmy do wypushowanego przed chwilą projektu Example1 prosty plik README, który posiadać będzie krótki opisu projektu.

Po kliknięciu w przycisk Add a Readme (znajdującego się na stronie głównej repozytorium Example1), zawartość tworzonego pliku wypełniłem w następujący sposób:

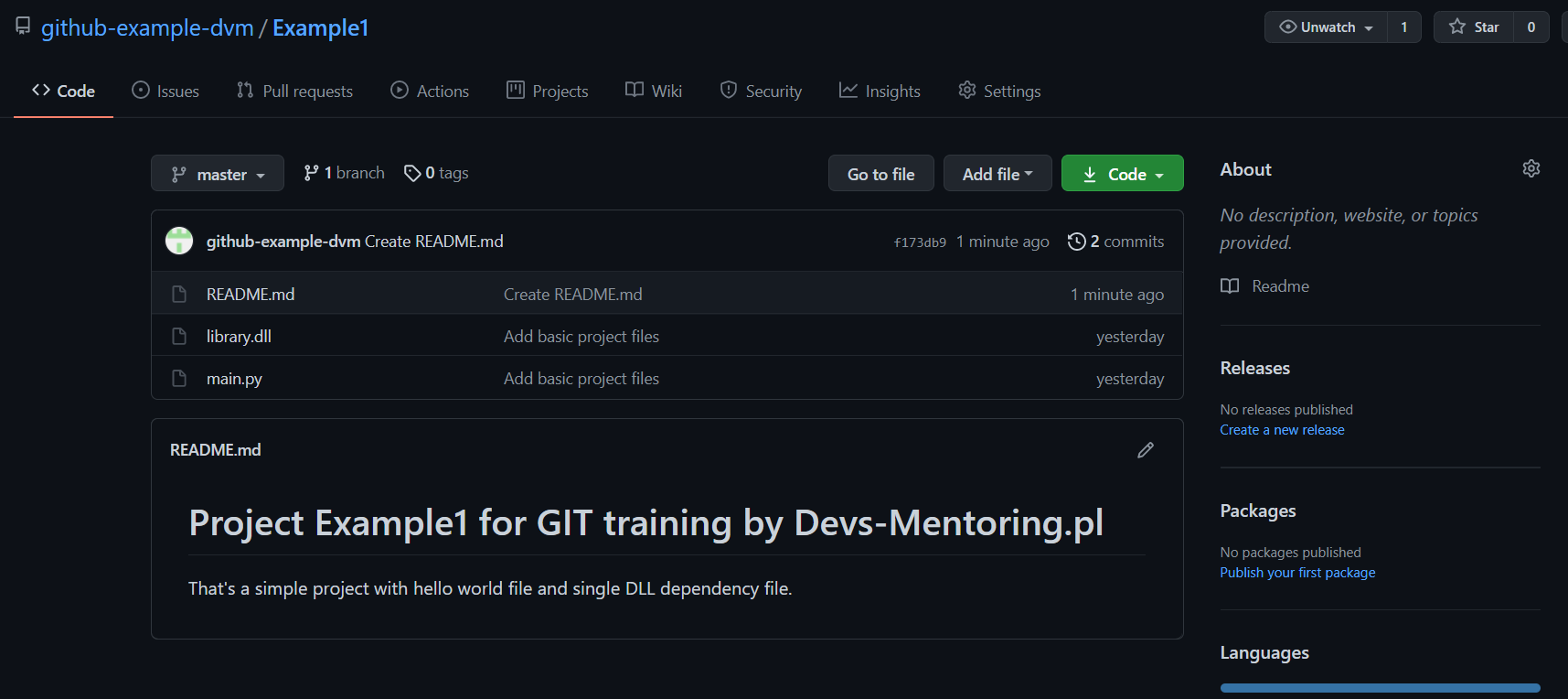


*Przykładowo wypełnione README*

Po czym zmianę zatwierdziłem poniższym przyciskiem:



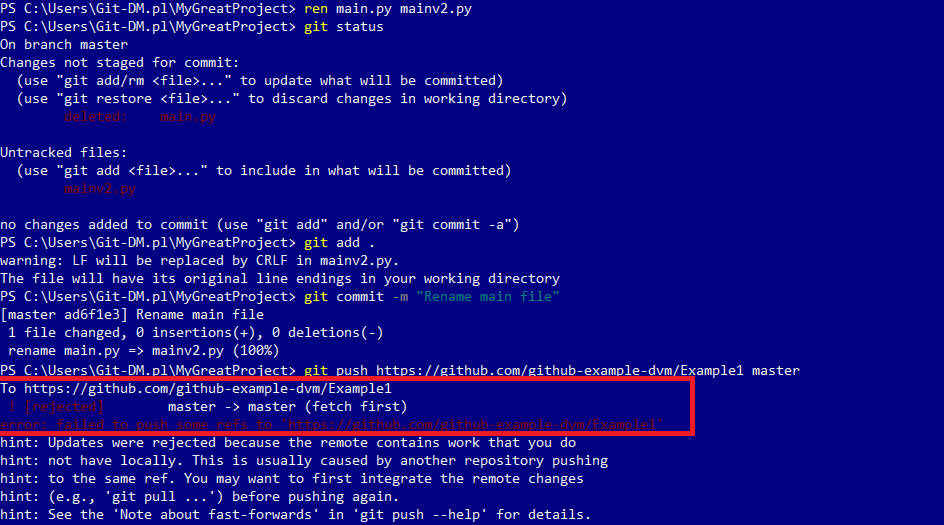
Od teraz widok główny repozytorium Example1 kształtuje się następująco:



Gdy będziesz tworzył rozbudowane projekty i zaczniesz zastanawiać się, jak napisać dobre README, to zapoznaj się z tym linkiem: [othneildrew/Best-README-Template: An awesome README template to jumpstart your projects!](https://github.com/othneildrew/Best-README-Template). Przedstawia on wzór pliku, na podstawie którego warto bazować, tworząc własny README.

**Update-owanie i klonowanie repozytorium z GH**

W poprzednim punkcie zmieniliśmy plik README z poziomu GH. Z powodu takiej zmiany musimy zaktualizować nasze lokalne repozytorium. A to dlatego, że nie jest ono zsynchronizowane z repozytorium umieszczonym na GH. Przez brak synchronizacji mam na myśli to, że **ostatni commit** wprowadzony w projekcie zdalnym nie jest taki sam, co ten, który znajduje się lokalnie. Nienaprawienie tego problemu, spowoduje problemy przy commitowaniu kolejnych zmian. Możesz spróbować na w dowolny sposób lokalnie zmodyfikować zawartość katalogu projektowego i wypushować te zmiany na GH. Ujrzysz wówczas następujący komunikat:

****

*Zmiana nazwy pliku main.py oraz próba wypushowania commita*

Komunikat mówi: “*failed to push some refs to <URL repozytorium>.*”. A to dlatego, iż *“(...) the remote contains work that you do not have locally (...)”.* Rozwiązanie tego problemu jest następujące: musimy zaktualizować lokalną wersję repozytorium o wprowadzone na GH zmiany. Umożliwi nam to polecenie **git pull**.

Git pull to kolejna komenda niezbędna dla każdego dewelopera w pracy z GIT-em. Umożliwia ona aktualizowanie katalogu o zmiany, które zostały wprowadzone wewnątrz zdalnego repozytorium. W ten sposób synchronizujemy obie wersje projektu.

Zauważ, że wykorzystanie tej komendy sprawdza się już na początku pracy z systemem kontroli wersji - konieczne jest jej użycie po, np. zmianie zawartości README. Idąc jednak krok dalej, git pull jest istotnie ważny w pracy z innymi deweloperami nad jednym repozytorium.

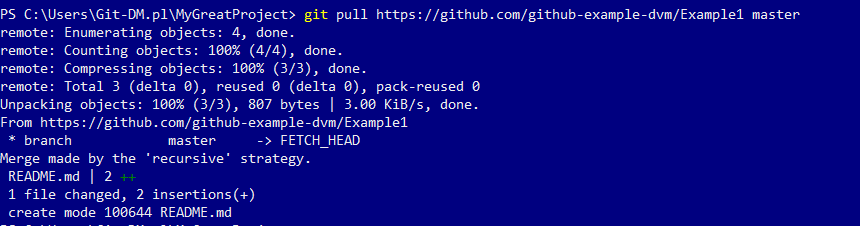
Wyobraź sobie, że dostęp do repozytorium Example1 ma również inny programista. W środę o godzinie 17stej, zakończył on prace nad nowym plikiem - feature.py - i wypushował taki commit na GH.

Teraz, gdybyśmy chcieli pobrać tak zauplodowany plik również do swojego katalogu roboczego, musimy wykorzystać właśnie proces pullowania i aktualizacji swojego projektu. W przeciwnym razie będziemy pracowali na starej wersji projektu, co może przysporzyć nam poznany już problem z commitowaniem własnych zmian.

Do tematu projektowej kolaboracji z innymi programistami wrócimy w dalszych szkoleniach, konieczne choćby będzie poruszenie tematu merge conflictów. Na ten moment jednak zapamiętaj zastosowanie i ideę wykorzystania polecenia git pull.

Składnia polecenia git pull:

git pull <url\_repozytorium> <gałąź\_projektu (np. master)>



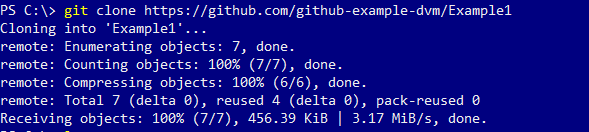
**Git clone**

Wyobraź sobie sytuację, w której zostałeś zmuszony wymienić swoją stację roboczą (na której do tej pory lokalnie programowałeś i budowałeś repozytorium) na inny sprzęt. Wówczas, chcąc kontynuować prace nad swoim projektem, konieczne będzie pobranie całego projektu z GitHub-a.

Git pull w tym przypadku się nie sprawdzi, ponieważ służy on do aktualizowania pobranej już wersji projektowej. Sytuacja, w której się znaleźliśmy wymaga przecież od nas zaciągnięcia kompletnego projektu. Wówczas aby to zrealizować, użyjemy polecenia git clone.

Pobierzmy repozytorium do dowolnego (innego niż dotychczas) miejsca na komputerze. W moim przypadku wykorzystam ścieżkę C:\.

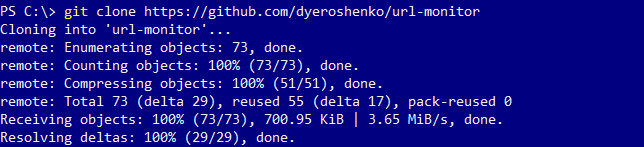
Chcąc zatem pobrać raz jeszcze cały katalog z repozytorium Example1 - do zupełnie niezależnego miejsca na swoim lokalnym komputerze - wpiszemy git clone <url\_repozytorium>.



*Klonowanie repozytorium od usera Example1*

Co ciekawe, nie musimy być właścicielem danego repozytorium, aby móc je sklonować. Polecenie git clone możemy tak naprawdę wykonać na dowolnym projekcie.

Poniższy przykład dobrze to pokazuje:



*Klonowanie repozytorium od usera dyeroshenko*

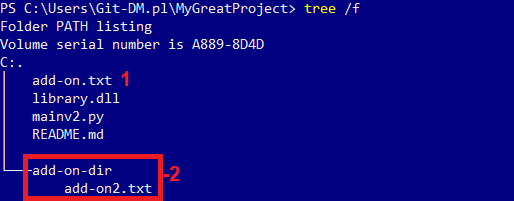
**Ignorowanie plików w repozytorium**

Czasami w naszej pracy może okazać się, że nie chcemy pushować wszystkich plików/katalogów z folderu roboczego do zdalnego repozytorium na GH. Takimi plikami mogą być często zasoby konfiguracyjne, ustawienia środowiska wirtualnego itp. A to dlatego, że udostępniając projekt publicznie, często chcemy umożliwić użytkownikowi jego pobranie i uruchomienie we własnym środowisku (oczywiście z odpowiednio zainstalowanymi modułami i zasobami). Dodatkowo, pushowane repozytorium powinno być maksymalnie lekkie i przenośne, dlatego udostępnianie go wraz z dodatkowymi zasobami jest zbędne.

Powinniśmy zatem stworzyć specjalny plik GIT-owy, który umożliwia zdefiniowanie, jakie pliki i katalogi mają być ignorowane i nie powinny być wrzucane do zdalnego repozytorium w procesie pushowania. Ten specjalny plik nosi nazwę .gitignore. Umieszczamy go wewnątrz repozytorium lokalnego i listujemy w nim nazwy plików i katalogów, które mają być ignorowane (dzięki temu będą one traktowane tak, jakby w ogóle nie znajdowały się w katalogu).

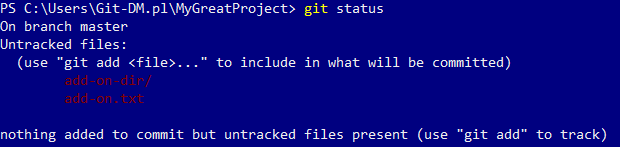
Umieśćmy w Naszym folderze z repozytorium lokalnym jeden dodatkowy plik: add-on.txt oraz katalog add-on-dir/, a w nim add-on2.txt.

Po modyfikacjach, katalog z projektem powinien wyglądać następująco:



Numerami zaznaczyłem nowo dodany katalog i pliki.

Teraz po wpisaniu polecenia git status, ujrzymy:



Wszystko zgodnie z oczekiwaniami - nowo wprowadzone zmiany znajdują się w pierwszym stanie - working directory.

**Problem:**

Załóżmy, że chcielibyśmy teraz wypushować tylko i wyłącznie plik add-on.txt. W związku z tym konieczne jest dodanie katalogu add-on-dir/ do zasobów ignorowanych - tak, aby plik mimo że fizycznie znajduje się wewnątrz katalogu, nie był rozpatrywany przez system kontroli wersji.   
  
1. Aby dodać katalog add-on-dir/ do ignorowanych, utwórzmy w pierwszej kolejności plik .gitignore.

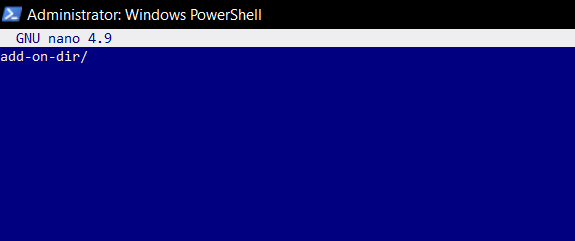
$ touch .gitignore

$ nano .gitignore

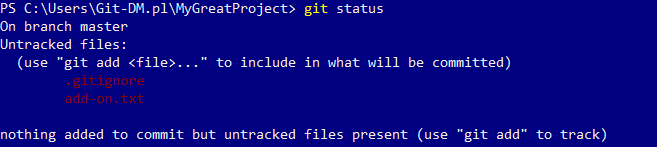
Polecenie nano spowoduje otwarcie edytora tekstu dla pliku .gitignore, w którym możemy umieszczać patterny nazw zasobów do zignorowania. Wzory umożliwiają nam ignorowanie, np. wielu plików jednocześnie, grupy zasobów o określonych rozszerzeniach itd.

Wskazówki, jak tworzyć takie wzory możesz znaleźć tutaj: [Ignoring Files and Directories in Git (.gitignore)](https://linuxize.com/post/gitignore-ignoring-files-in-git/)

My akurat wykorzystamy najprostszy z możliwych wzór - w .gitginore podamy bezpośrednio nazwę katalogu, który chcemy zignorować:



Po utworzeniu i zapisaniu zawartości w takim pliku, wpiszmy git status:



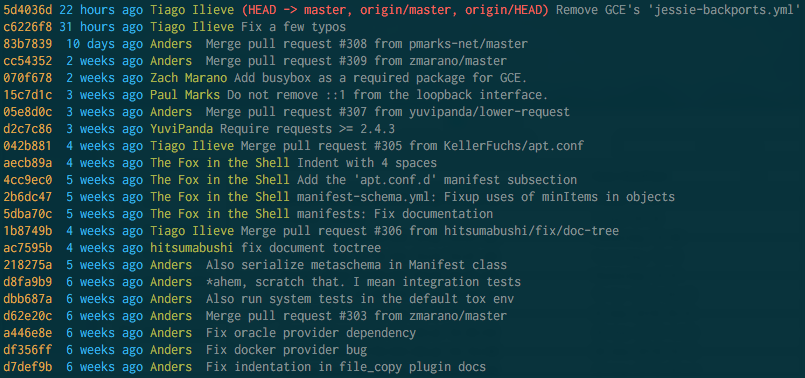
Zauważ, że od tego momentu, katalog add-on-dir/ przestał być śledzony i jest dla systemu kontroli wersji całkowicie niewidoczny. To jest właśnie użyteczność i przydatność .gitignore.

Teraz możesz śmiało wypushować do repozytorium kolejny commit, który będzie bazował na dodaniu pliku .gitignore wraz z plikiem testowym add-on.txt.

**Kontrolowanie historii repozytorium**

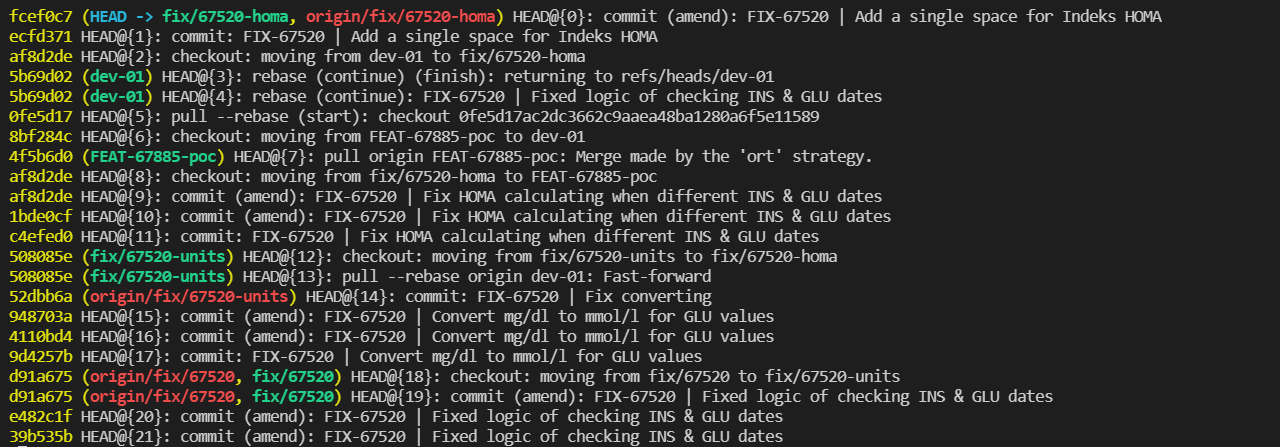
W trakcie pracy z GIT-em na pewno niejednokrotnie będziesz chciał skontrolować historię dotąd wprowadzonych zmian w swoim repozytorium. Oczywiście, tak jak już zdążyłeś się przekonać, historię commitów możesz łatwo sprawdzić z poziomu GUI GitHuba, jednak możliwe jest również zweryfikowanie historii commitów (na danym branchu) z poziomu lokalnego projektu. A to wszystko dzięki poleceniu: **git log.**

Poniżej przykładowy output polecenia, w którym wyświetlone są liczne commity z uwzględnieniem użytkowników, którzy taką operację przeprowadzili (na screenie widać, że nad naszym repozytorium pracuje wielu developerów):



Źródło: Stack Overflow

Drugim, również użytecznym, ale niosącym nieco bardziej szczegółowe informacje poleceniem jest: **git reflog**. Komenda ta przedstawia nam historię wszystkich zmian i operacji przeprowadzonych na **lokalnym repozytorium**. Tak więc wyświetla ona wszystkie modyfikacje, jak np. zmiana nazw branchy, przechodzenie między branchami, resetowanie zmian, pullowanie wersji (są to już nieco bardziej zaawansowane polecenia, które poznamy w następnym szkoleniu).



Przykładowy output polecenia git reflog

**Podsumowanie**

W szkoleniu tym poznaliśmy podstawowy workflow z poleceniami GIT-a. Przez użycie takich poleceń jak git status, git init, git add, git commit, git push, pokazałem Ci, jak udostępniać dowolny lokalny projekt do publicznego repozytorium. Przedstawiłem ci również ideę pliku .gitignore oraz jak sposób na jego poprawne utworzenie. W szkoleniu tym pojawiło się również wyjaśnienie takich zagadnień, jak: origin, master, main. I wreszcie dowiedziałeś się, czym jest pullowanie i klonowanie repozytorium oraz w jaki sposób przeprowadzać te operacje.   
  
Wszystkie te informacje umożliwią Ci podstawowe zarządzanie swoimi repozytoriami. Aczkolwiek, chcąc szlifować swoje GIT-owskie umiejętności, nie możemy bazować jedynie na tych wiadomościach. Dlatego zapraszam Cię do kolejnych szkoleń, gdzie poruszymy różne kruczki i bardziej szczegółowo skupimy się na pracy z omawianym systemem kontroli wersji.

**Dodatek**

Na koniec i podsumowanie tego szkolenia przesyłam, tzw. GIT Cheatsheet, który zawiera wszystkie najczęściej wykorzystywane polecenia przy pracy z GIT-em. Dużą część tych poleceń jeszcze nie omówiliśmy, będziemy je omawiali progresywnie. Jednak już teraz jest to dobry moment na pobranie i zapisanie u siebie na dysku niżej przedstawionego screenu (który znajdziesz również pod tym linkiem: [Post Devs-Mentoring.pl](https://www.facebook.com/devs.mentoring/photos/a.129929668863424/230863322103391/) )

****