

# Kontrola wersji w Git

v3.1

# Plan

1. Nauka GIT
2. Wprowadzenie
3. Uruchamianie terminala
4. Instalacja Gita
5. Jak działa kontrola wersji?
6. Podstawowa składnia Gita
6. Praca zespołowa
7. Konflikt!
8. Problemy z Git!
9. Branche, czyli odgałęzienia kodu
10. Plik .gitignore
11. Przydatne linki dla użytkowników Gita
12. Obowiązkowa informacja o prawach autorskich

# Nauka GIT

# Nauka GIT

**Przed kursem musicie dobrze poznać system kontroli wersji GIT. Będziecie go używać do ściągania zadań na zajęciach jak i do wysyłania swoich odpowiedzi żeby mentor mógł je sprawdzić.**

Gita będziemy poznawać stopniowo na początek zajrzyj do poniższych źródeł:

- <https://git-scm.com/book/pl/v1> - książka twórców GITa. Tłumaczy działanie całego systemu.
- <https://www.git-tower.com/learn/git/ebook> - wprowadza w używanie GITa od podstaw.

**Jeżeli nadal masz czas i chcesz więcej dowiedzieć się o systemie GIT możesz obejrzeć -**  
**[https://www.youtube.com/watch?v=SWYqp7iY\\_Tc](https://www.youtube.com/watch?v=SWYqp7iY_Tc)**

# Wprowadzenie

# Kontrola wersji – co to znaczy?

System kontroli wersji (ang. version control system, VCS) to oprogramowanie służące do śledzenia zmian np. w kodzie źródłowym programu.

Istnieje wiele systemów kontroli wersji (m.in. CVS, Subversion, Mercurial, Perforce).

Najbardziej rozpowszechnionym i popularnym VCS używanym przez branżę IT jest **Git**.



# Dlaczego używamy kontroli wersji?

Używanie systemu kontroli wersji jest bardzo ważne przy tworzeniu projektu. Osiągamy dzięki temu rozwiązaniu następujące korzyści:

- **Współpraca wielu osób** – systemy kontroli wersji wspomagają łatwą pracę nad jednym projektem. Rozwiązują konflikty (czyli sytuacje, w których kilka osób pracuje na jednym pliku).
- **Pomoc w organizacji wersji projektu** – dzięki możliwości nadawania tagów i cofania naszego kodu do nich jesteśmy w stanie trzymać wszystkie wersje swojego programu.
- **Cofanie wprowadzanych zmian** – możemy przywrócić nasz kod do dowolnego punktu w przeszłości.
- **Trzymanie backupów naszych projektów** – dzięki użyciu zewnętrznych repozytoriów będziemy mieli dostęp do naszego kodu praktycznie z każdego komputera. Nawet jeżeli nasz ulegnie awarii.

# Co to jest Git?

Git jest najpopularniejszym systemem kontroli wersji. Stworzony w 2005 roku przez Linusa Torvaldsa (twórcę Linuxa) jako system kontroli do wspomagania projektów Open Source.

Jest typowym systemem rozproszonym. Oznacza to, że pełna historia projektu znajduje się na każdym komputerze, który ma repozytorium z tym projektem.

Git bardzo dobrze sobie radzi z wszelkimi plikami tekstowymi, gorzej z plikami binarnymi (zdjęciami, plikami pdf, itp).





# Uruchamianie terminala

# Jak otworzyć terminal?

## Jeśli jesteś użytkownikiem Ubuntu

Otwórz terminal następującą kombinacją klawiszy:

**Ctrl+Alt+T**



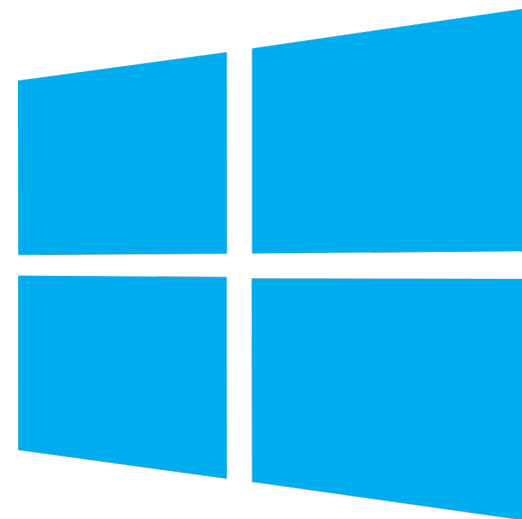
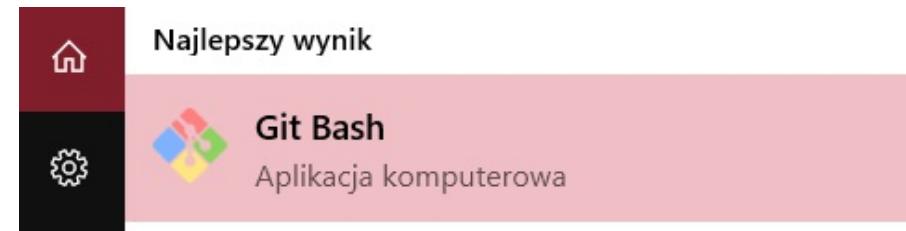
Powinno ukazać się coś na wzór tej linii w Twoim terminalu:

```
marcin@xwing:~$
```

# Jak otworzyć terminal?

## Jeśli jesteś użytkownikiem Windowsa

1. Zainstaluj Git Basha (instrukcja w następnym rozdziale).
2. Otwórz Git Basha.



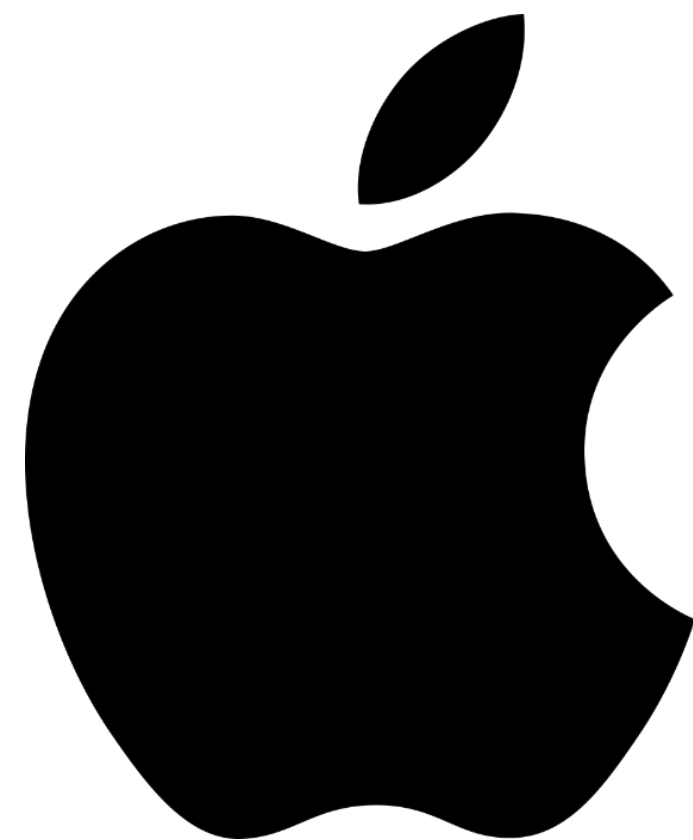
Powinno ukazać się coś na wzór tej linijki w Twoim terminalu:

```
Agata@Agata-Coderslab MINGW64 ~$
```

# Jak otworzyć terminal?

## Jeśli jesteś użytkownikiem MacOS

1. Wciśnij kombinację klawiszy **Cmd+spacja**.
2. W oknie **Spotlight** zacznij wpisywać **terminal**.
3. Potwierdź klawiszem **Enter**.



Powinno ukazać się coś na wzór tej linii w Twoim terminalu:

```
MacBook-Pro: ~SomeUser
```

# Instalacja Gita

# Instalacja Gita

## Jeśli jesteś użytkownikiem Ubuntu

1. Otwórz terminal.
2. W terminalu wpisz następujące komendy:

```
sudo apt-get update  
sudo apt-get install git
```

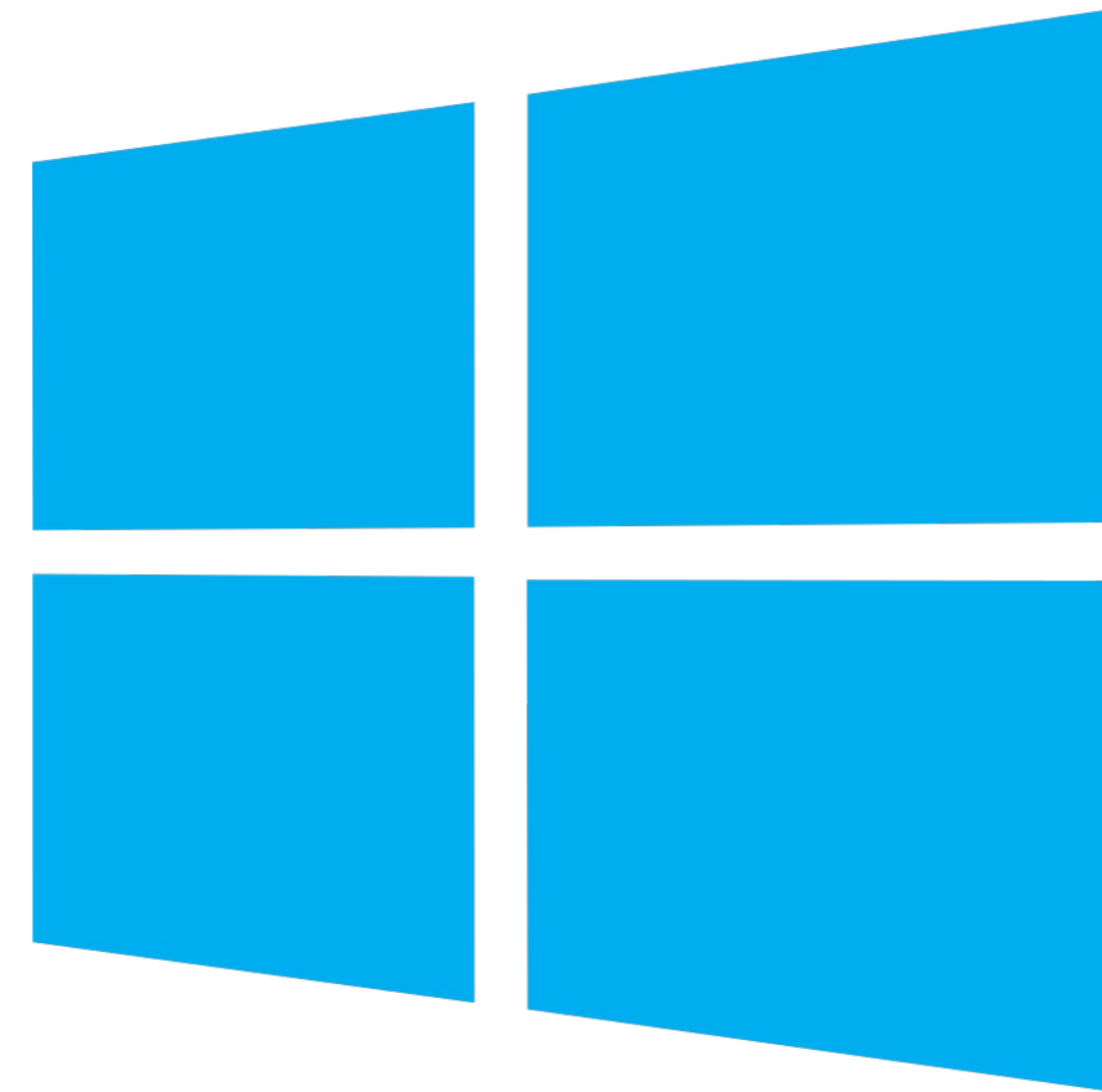
Osoby korzystające ze skryptu instalacyjnego dostarczonego przez nas mają już zainstalowanego Gita.



# Instalacja Gita

## Jeśli jesteś użytkownikiem Windowsa

1. Wejdź na stronę <https://git-scm.com/download/win> (program zacznie pobierać się automatycznie).
2. Podczas instalacji wybieraj proponowane opcje.
3. Uruchom pobrany program.



# Instalacja Gita

## Jeśli jesteś użytkownikiem MacOS

1. Wejdź na stronę <https://git-scm.com/download/mac> (program zacznie pobierać się automatycznie).
2. Podczas instalacji wybieraj proponowane opcje.
3. Uruchom pobrany program.

Osoby korzystające ze skryptu instalacyjnego dostarczonego przez nas mają już zainstalowanego Gita.





# Konfiguracja Gita

## Podstawowe ustawienia narzędzia

Git potrzebuje Twoich danych, by dodawać je do informacji o zmianach w kodzie. W tym celu dodaj do konfiguracji Twoje imię i nazwisko oraz email.

## Uruchom terminal i wpisz następujące komendy:

```
git config --global user.name "Imię i nazwisko"  
git config --global user.email "adres email"
```

# Konfiguracja Gita

## Edytor komunikatów

Czasami zdarza się, że musisz wpisać komentarz do zmian interaktywnie, przy użyciu edytora tekstu. Standardowym edytorem tekstu, używanym przez Gita jest **Vim**\*. Jeśli chcesz zmienić edytor\*\*, wykonaj instrukcje z kolejnych slajdów.

\* **Vim** to jeden ze starszych i podstawowych edytorów tekstów dla systemów UNIX i podobnych. Bardzo użyteczny, ale trudny w obsłudze, zwłaszcza dla początkujących. (<https://pl.wikipedia.org/wiki/Vim>)

\*\* Zaufaj nam.



# Konfiguracja Gita

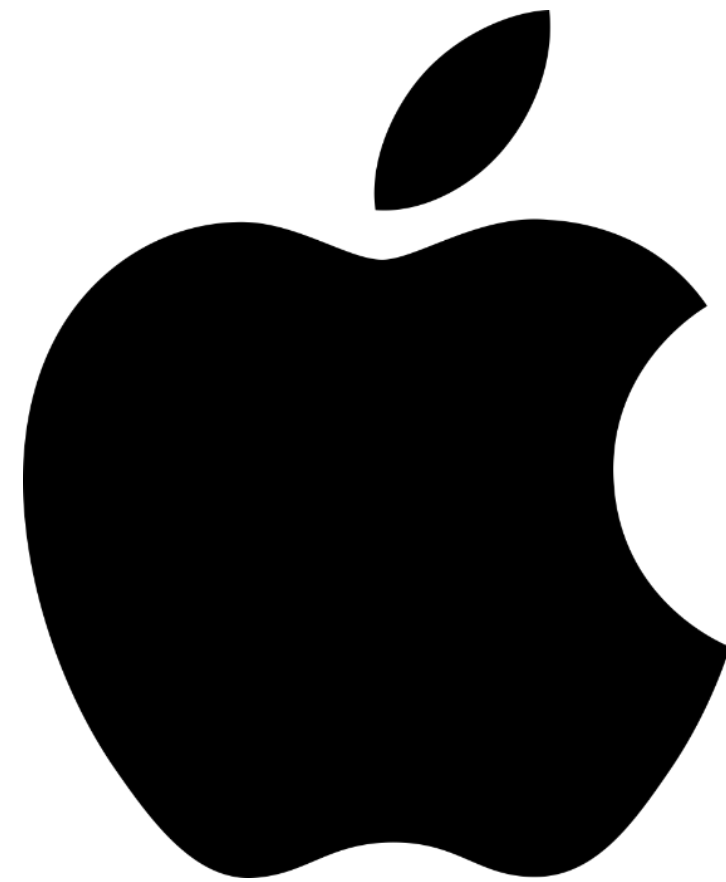


**Nano** jest wygodnym edytorem tekstu, uruchamianym w terminalu. Więcej o Nano [https://pl.wikipedia.org/wiki/Nano\\_\(program\)](https://pl.wikipedia.org/wiki/Nano_(program))).

**Ubuntu - otwórz terminal i wpisz:**

```
git config --global core.editor nano
```

# Konfiguracja Gita

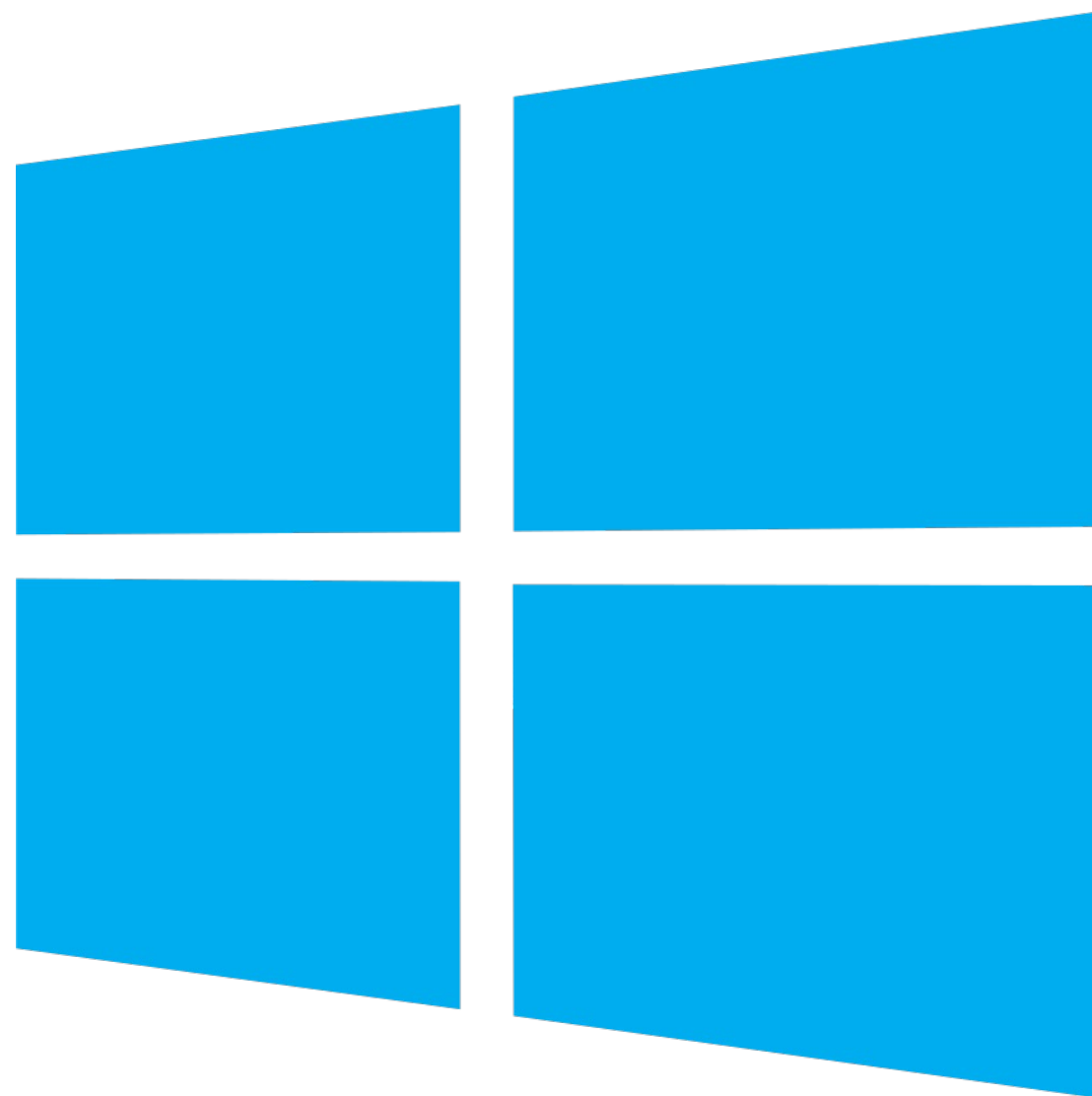


**Nano** jest wygodnym edytorem tekstu, uruchamianym w terminalu. Więcej o Nano [https://pl.wikipedia.org/wiki/Nano\\_\(program\)](https://pl.wikipedia.org/wiki/Nano_(program))).

**Mac - otwórz terminal i wpisz:**

```
git config --global core.editor nano
```

# Konfiguracja Gita



## Windows - otwórz terminal

Komendy te nie zwracają żadnych informacji (po wciśnięciu **Enter** w terminalu pojawi się nowa linia, w której możesz wpisać następną komendę).

Wpisz komendy (po wpisaniu każdej wciśnij **Enter**):

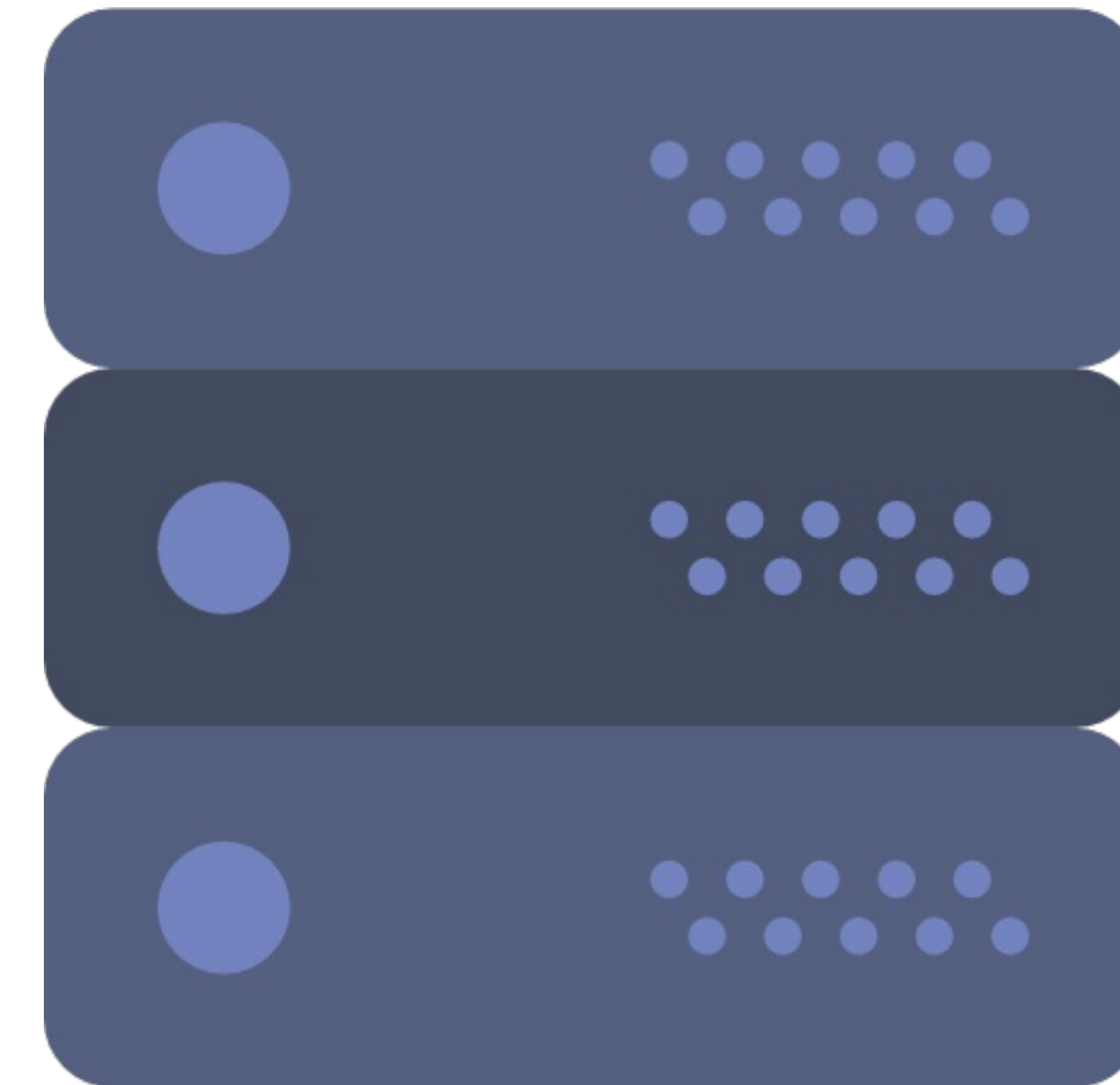
```
git config --global core.editor notepad  
git config --global format.commitMessageColumns
```

**Jak działa  
kontrola  
wersji?**

# Jak działa kontrola wersji?

Wyobraź sobie system kontroli wersji jako bazę danych. Gdy piszesz program, w każdym momencie możesz zapisać do tej bazy aktualny stan swojego kodu.

Gdy później sprawdzisz ten stan (nazywajmy go od tego momentu wersją), oprogramowanie pokaże Ci, czym aktualna wersja różni się od poprzedniej. Dzięki temu możesz prześledzić, co zmieniło się w Twoim kodzie.



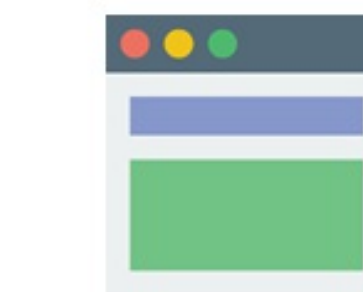
# Jak działa kontrola wersji?

## Poniedziałek, 9:00

Agata tworzy następujące nowe pliki:

- **index.html**
- **o-nas.html**

Umieszcza je w systemie kontroli wersji i opisuje zmianę.



index.html



o-nas.html



Poniedziałek, 9.00:

- index.html (nowy),
- o-nas.html (nowy).





# Jak działa kontrola wersji?

Środa, 14:00

Agata modyfikuje następujący plik:

➤ **index.html**

Agata tworzy dwa nowe pliki:

➤ **regulamin.html**

➤ **smieszny-kotek.png**

Umieszcza je w systemie kontroli wersji i opisuje zmianę.



index.html



regulamin.html



smieszny-kotek.png



Środa, 14.00:

- index.html (modyfikacja),
- regulamin.html (nowy),
- smieszny-kotek.png(nowy).

Poniedziałek, 9.00:

- index.html (nowy),
- o-nas.html (nowy).

# Jak działa kontrola wersji?

**Piątek, 16.55 (do weekendu pięć minut)**

Marcin modyfikuje plik:

➤ **regulamin.html**

Z pliku **regulamin.html** usuwa większość treści i zmienia całkowicie wygląd.

Umieszcza je w systemie kontroli wersji i opisuje zmianę.



**Piątek, 16:55:**

- regulamin.html (modyfikacja)

**Środa, 14.00:**

- index.html (modyfikacja),
- regulamin.html (nowy),
- smieszny-kotek.png(nowy).

**Poniedziałek, 9.00:**

- index.html (nowy),
- o-nas.html (nowy).

# Jak działa kontrola wersji?

## WTEM!

Okazuje się, że Marcin niepotrzebnie zmodyfikował plik **regulamin.html**!

Czy cała praca nad tym plikiem poszła na marne?

**Na szczęście nie!** Marcin może wrócić do wersji sprzed zmiany, czyli wersji 2., ze środy, z godziny 14.00.

Weekend uratowany. Uff...



~~Piątek, 16:55:~~

~~• regulamin.html (modyfikacja)~~

**Środa, 14.00:**

- index.html (modyfikacja),
- regulamin.html (nowy),
- smieszny-kotek.png(nowy).

**Poniedziałek, 9.00:**

- index.html (nowy),
- o-nas.html (nowy).

# **Podstawowa składnia Gita**

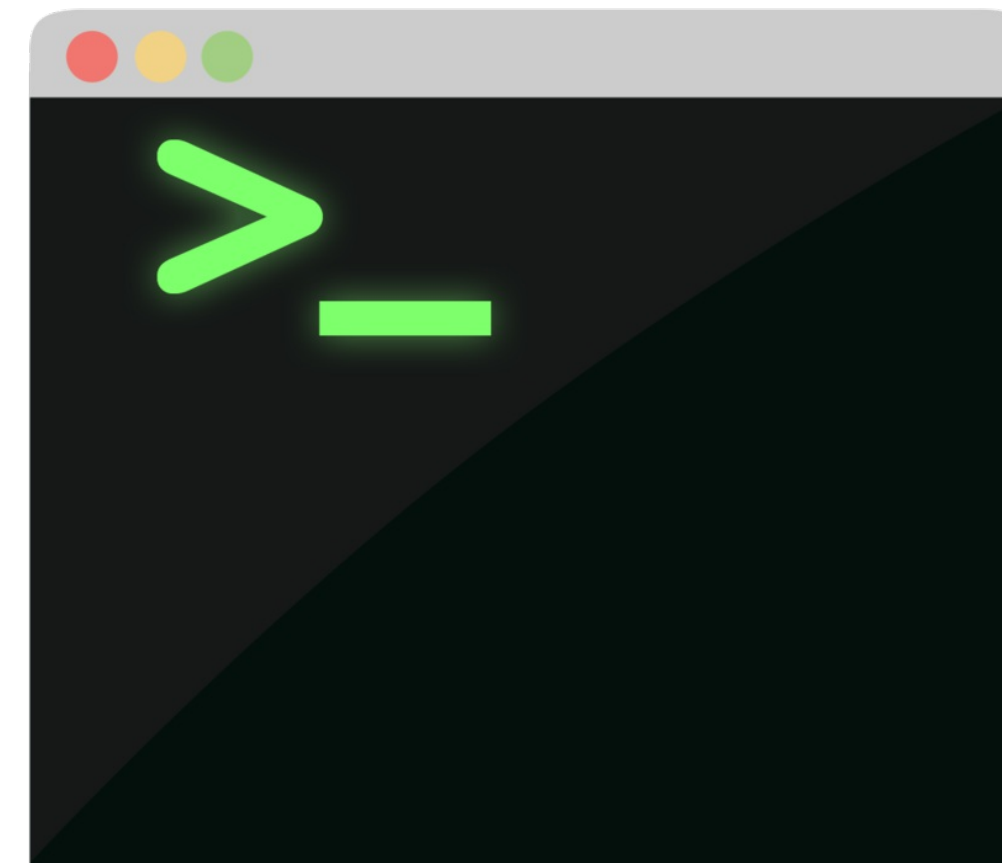


# Podstawy korzystania z Gita

Z Gita możemy korzystać na dwa różne sposoby:

- przez program konsolowy,
- przez graficzny interfejs użytkownika (trzeba go osobno doinstalować – istnieje wiele różnych programów).

Podczas naszego kursu polecamy korzystanie z komend konsolowych Gita. Dzięki temu poznacie sposób jego działania i dowiedziecie się, co dokładnie można z nim zrobić.



# Komendy Gita

Najłatwiej nauczyć się komend Gita dzięki korzystaniu z nich. Przejdź zatem interaktywny tutorial, który możesz znaleźć na stronie:

<https://try.github.io/levels/1/challenges/1>

Podczas przerabiania tego tutorialu zapisuj sobie, co robią poszczególne komendy. Te notatki bardzo Ci się przydadzą.



# GIT CheatSheet

Możesz też skorzystać z gotowego zbioru najprzydatniejszych komend. Jest to tak zwany **Git CheatSheet**.

Możesz znaleźć wiele takich wersji np.:

- <https://www.git-tower.com/blog/git-cheat-sheet>
- <https://services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf>
- <http://www.cheat-sheets.org/saved-copy/git-cheat-sheet.pdf>



# Repozytoria



# Repozytorium

Git przechowuje Twój kod źródłowy w tzw. **repozytorium**.

Są to zapisane kolejne zmiany w Twoim kodzie (od samego początku pracy), które nawarstwiają się od samego początku pracy nad projektem aż do aktualnej wersji.

Repozytorium może być **lokalne** lub **zdalne**.

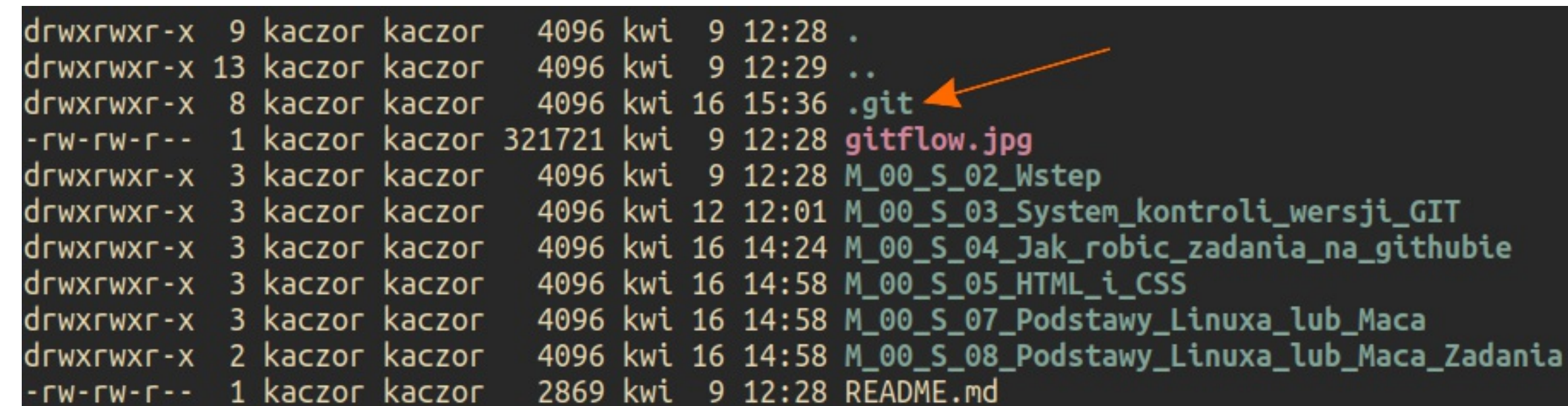


# Repozytorium lokalne

**Repozytorium lokalne** (local repository) znajduje się na Twoim komputerze. Podczas pracy nad swoim programem możesz (a nawet powinieneś/powinnaś) umieszczać w nim wszystkie zmiany.

**Repozytorium lokalne** znajduje się w głównym katalogu Twojego projektu w ukrytym folderze o nazwie **.git**.

```
drwxrwxr-x  9 kaczor kaczor  4096 kwi  9 12:28 .
drwxrwxr-x 13 kaczor kaczor  4096 kwi  9 12:29 ..
drwxrwxr-x  8 kaczor kaczor  4096 kwi 16 15:36 .git
-rw-rw-r--  1 kaczor kaczor 321721 kwi  9 12:28 gitflow.jpg
drwxrwxr-x  3 kaczor kaczor  4096 kwi  9 12:28 M_00_S_02_Wstep
drwxrwxr-x  3 kaczor kaczor  4096 kwi 12 12:01 M_00_S_03_System_kontroli_wersji_GIT
drwxrwxr-x  3 kaczor kaczor  4096 kwi 16 14:24 M_00_S_04_Jak_robic_zadania_na_githubie
drwxrwxr-x  3 kaczor kaczor  4096 kwi 16 14:58 M_00_S_05_HTML_i_CSS
drwxrwxr-x  3 kaczor kaczor  4096 kwi 16 14:58 M_00_S_07_Podstawy_Linuxa_lub_Maca
drwxrwxr-x  2 kaczor kaczor  4096 kwi 16 14:58 M_00_S_08_Podstawy_Linuxa_lub_Maca_Zadania
-rw-rw-r--  1 kaczor kaczor  2869 kwi  9 12:28 README.md
```

A terminal window with a dark background and light-colored text. It displays the output of a command, likely 'ls -la', showing a list of files and directories. The entries include permissions, owner, group, size, date, and filename. An orange arrow points from the right side of the terminal to the '.git' directory entry.

# Praca lokalna

## Nowy projekt Marcina - landing page

Marcin tworzy **repozytorium lokalne** (na razie puste). Żeby to zrobić wpisuje kod:

```
git init
```



(brak plików)

Powinien dostać taką odpowiedź:

```
marcin@xwing: ~/workspace/landing-page$ git init
Initialized empty Git repository in /landing-page/.git/
marcin@xwing: ~/workspace/landing-page$
```

# Praca lokalna

## Marcin zaczyna kontrolować wersje pliku

Marcin tworzy nowy plik: **landing-page.html**

Teraz Marcin musi poinformować Gita, że chce objąć ten plik kontrolą wersji:

```
git add landing-page.html
```

Komenda ta nie zwraca żadnej informacji.



(brak plików)



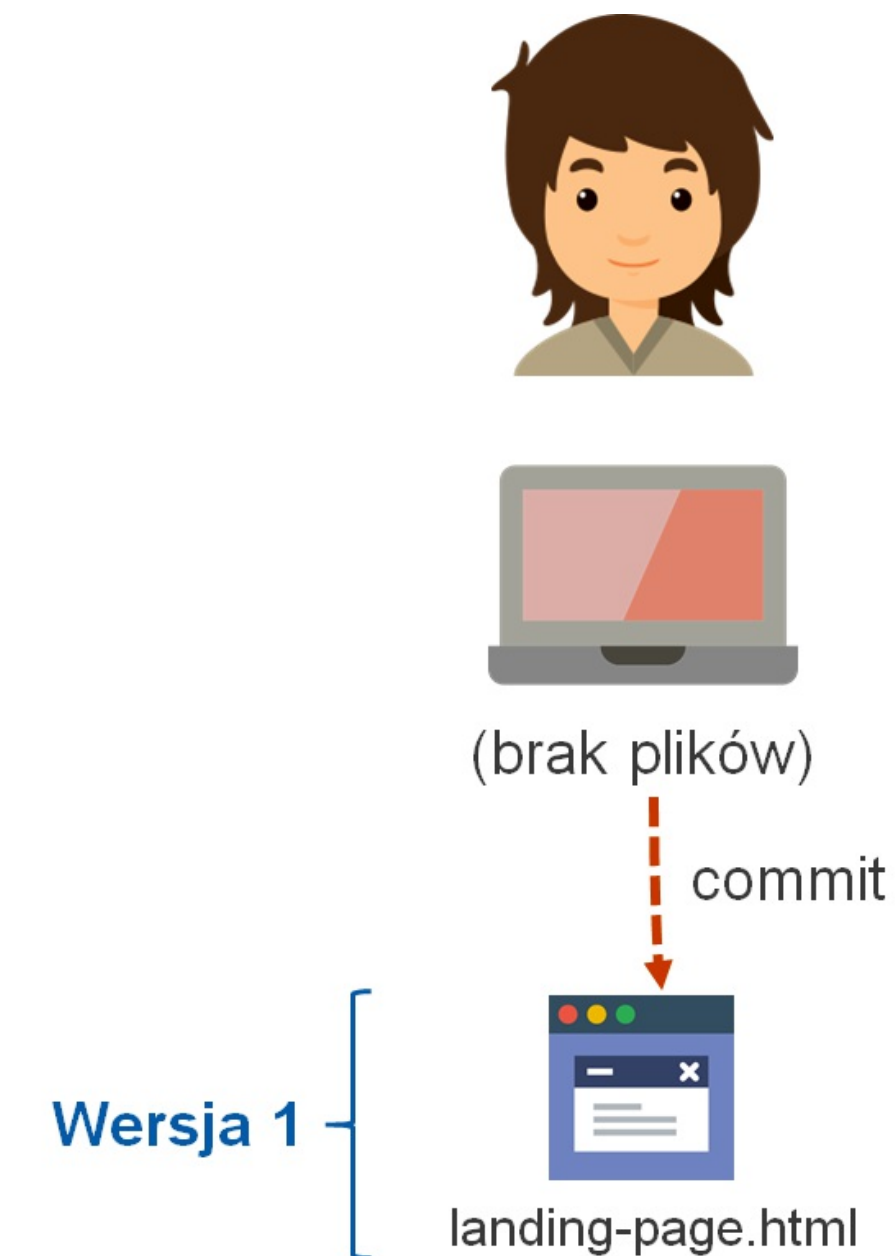
landing-page.html

# Praca lokalna

## Marcin umieszcza zmiany w systemie kontroli wersji

Marcin umieszcza zmiany w Gicie: **landing-page.html**

Marcin musi pamiętać, aby opisać swoje zmiany!



```
git commit -m "Dodałem nagłówek i stopkę"
```

Opcja **-m** oznacza, że Marcin chce opisać zmiany w plikach. Opis podaje w cudzysłowach, zaraz po parametrze. Ta informacja jest obowiązkowa! Git nie pozwoli zapisać zmian bez opisu!



# Repozytorium zdalne

**Repozytorium zdalne** (remote repository) znajduje się na zewnętrznym serwerze, np. **github.com**.

**Repozytorium zdalnego** używasz, gdy chcesz udostępnić swój kod innym (np. współpracownikom) lub przyłączyć się do istniejącego projektu.

# Repozytorium zdalne vs lokalne

Jeśli chcesz dołączyć do istniejącego projektu, skopiuj zdalne repozytorium na swój dysk. Ta czynność nazywa się **klonowaniem (clone)** repozytorium.

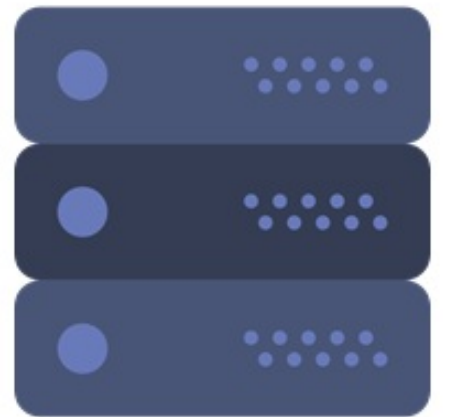
Powstaje wtedy lokalne repozytorium z aktualną wersją kodu.

Po wprowadzeniu zmian, umieszczasz je w lokalnym repozytorium (**commit**), a potem wypychasz (**push**) na serwer zdalny. Od tej pory inni programiści widzą Twoje zmiany.

# Praca na repozytoriach

## Nowy projekt - sklep internetowy

1. Agata tworzy **zdalne repozytorium** (na razie puste).



(brak plików)



# Praca na repozytoriach

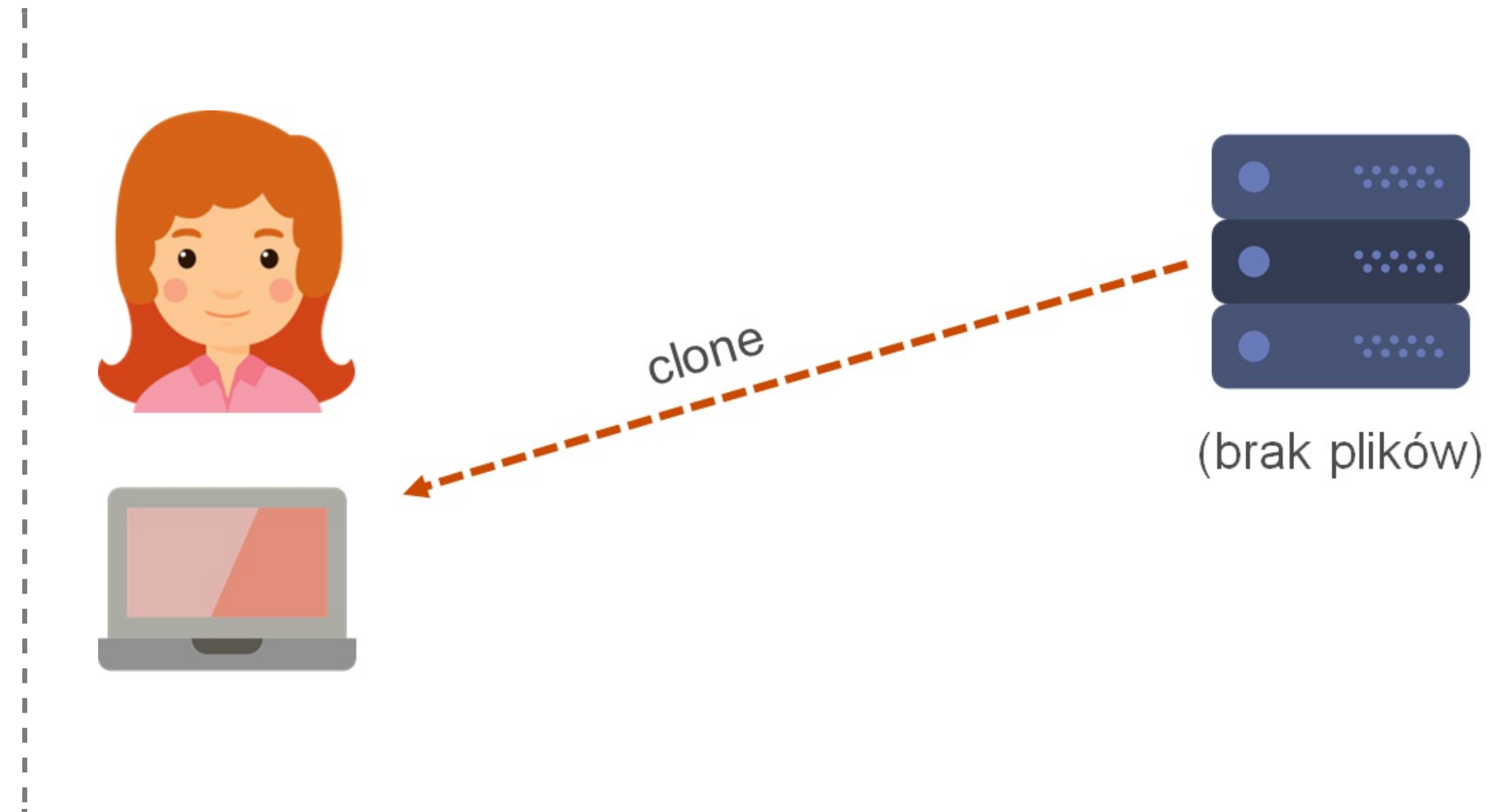
## Nowy projekt - sklep internetowy

2. Następnie Agata klonuje zdalne repozytorium do swojego laptopa, tworzy się **repozytorium lokalne**.

```
git clone <adres repozytorium>
```

W naszym przykładzie repozytorium ma nazwę: **sklep-internetowy.git**

```
git clone https://github.com/marcin-barylka/sklep-internetowy.git
Cloning into "sklep-internetowy"...
warning: You appear to have cloned an empty repository.
Checking connectivity... done.
```



# Praca na repozytoriach

## Agata tworzy nowy projekt - e-sklep

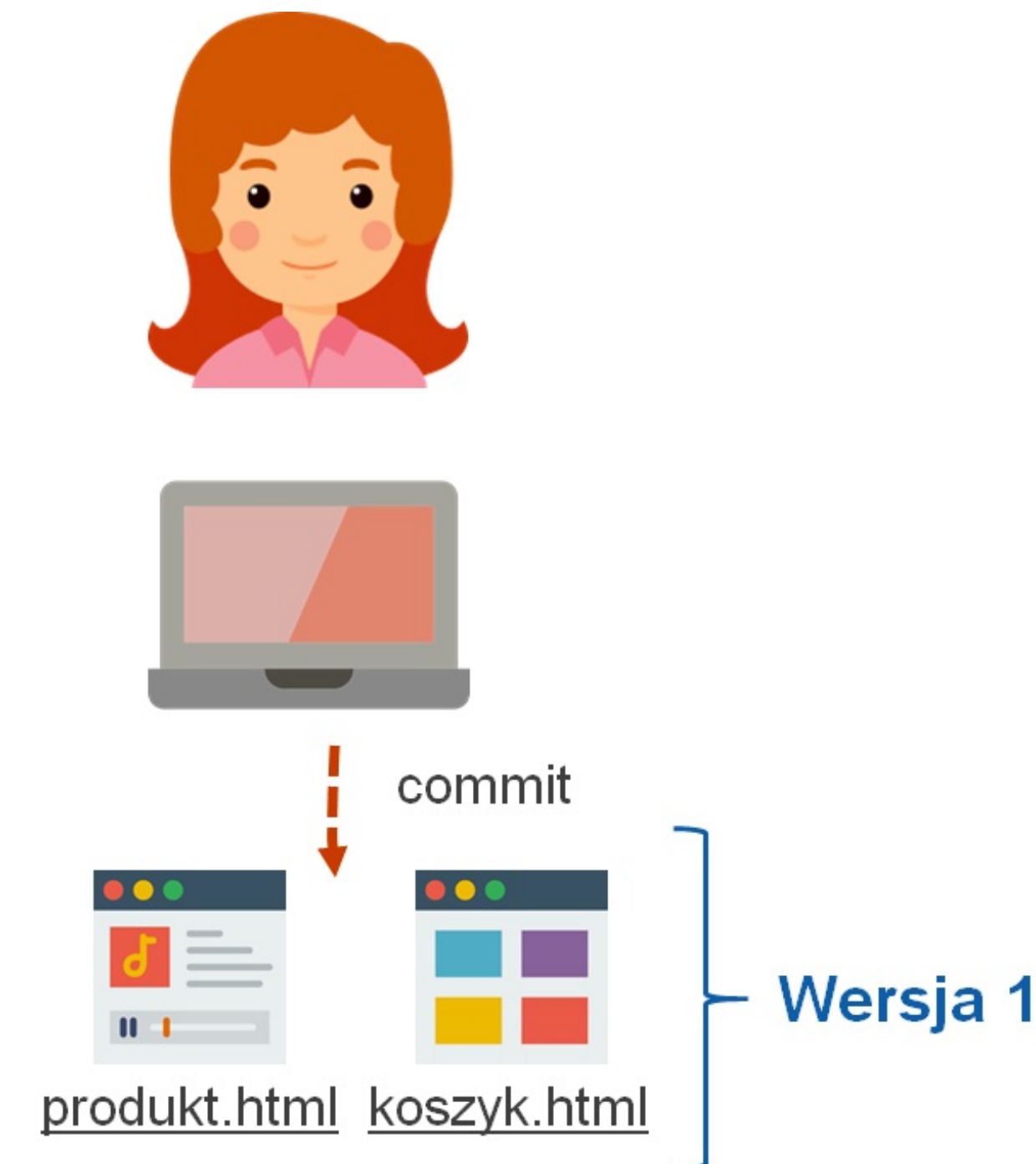
3. Agata pracuje nad nowymi plikami.

- **produkt.html**
- **koszyk.html**

Agata dodaje je do kontroli wersji:

```
git add *.html
```

**Te zmiany są widoczne tylko dla Agaty!**



Następnie Agata zapisuje zmiany na Git:

```
git commit -m "Strony produktu i koszyka"
```

# Praca na repozytoriach

## Aktualizacja zdalnego repozytorium

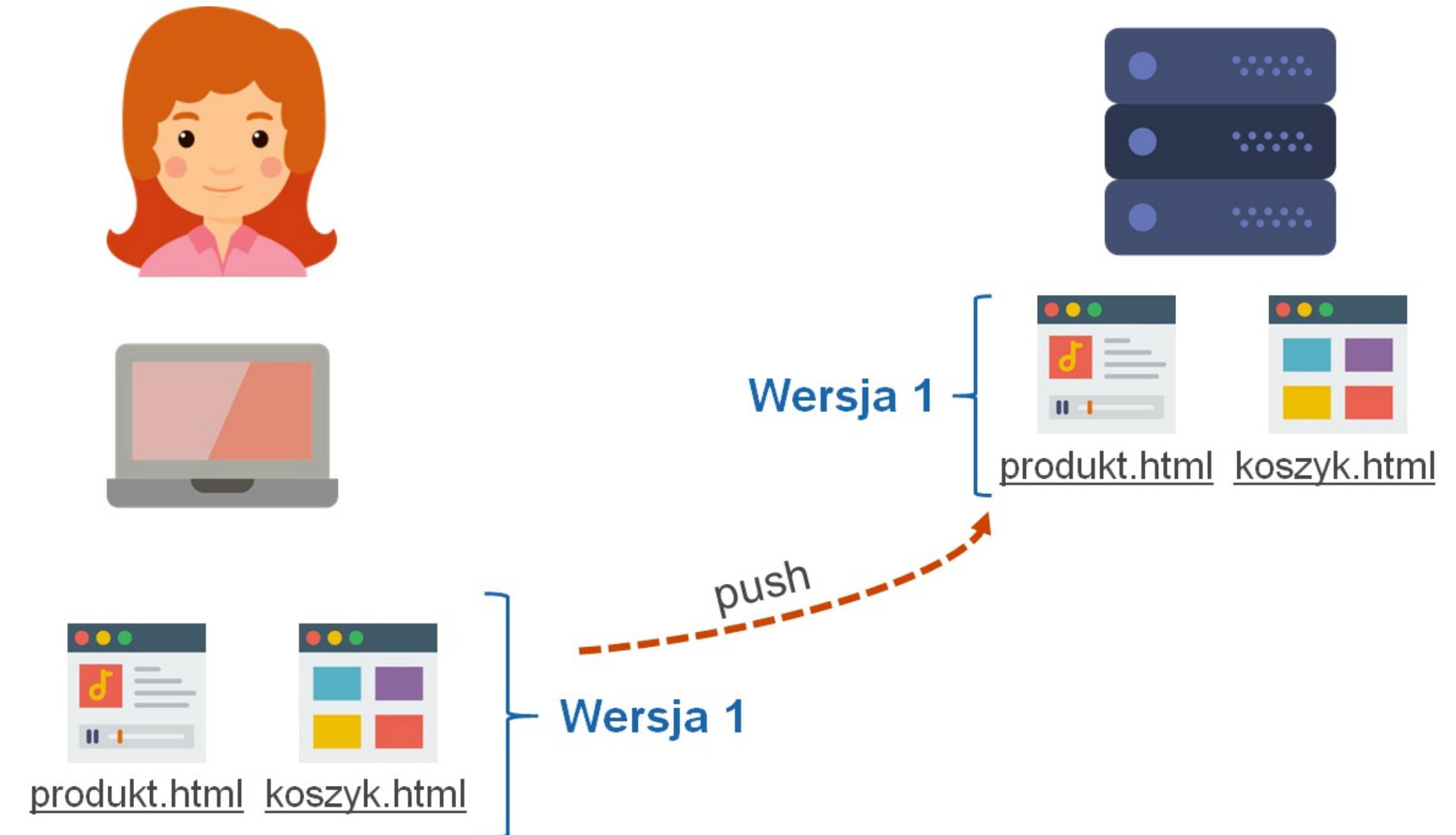
4. Agata wypycha ( **push**) swoje zmiany do zdalnego repozytorium.

➤ **produkt.html**

➤ **koszyk.html**

```
git push
```

Od tej pory jej zmiany są widoczne dla reszty programistów w zespole.



# Praca zespołowa

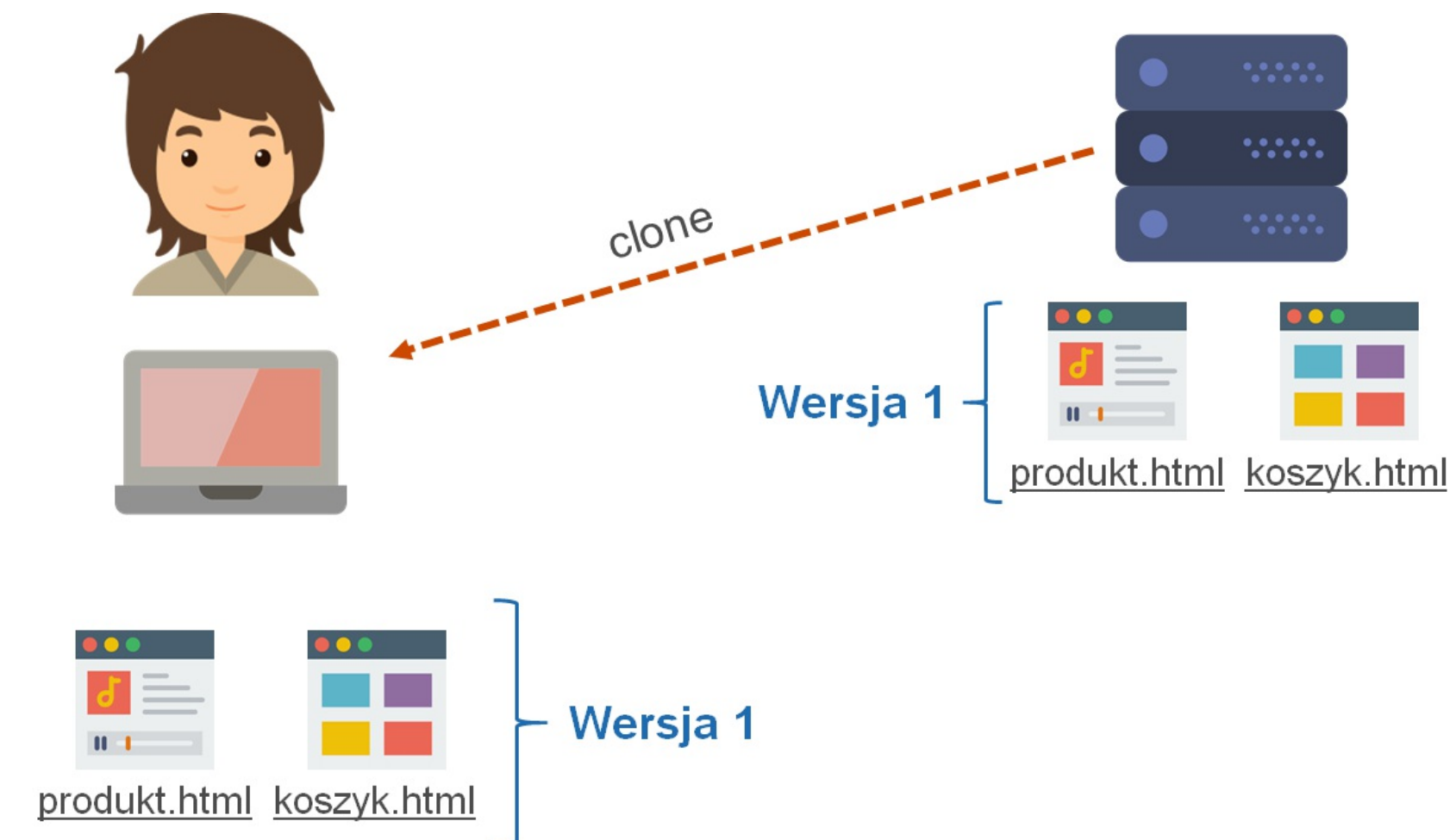
# Praca zespołowa

## Marcin dołącza do projektu

1. Marcin klonuje zdalne repozytorium do swojego laptopa, tworzy się **repozytorium lokalne**.

```
git clone <adres repozytorium>
```

2. Marcin pracuje nad stroną produktu (**produkt.html**) i stroną zamówienia (**zamowienie.html** - nowy plik, którego jeszcze nie ma w repozytorium).





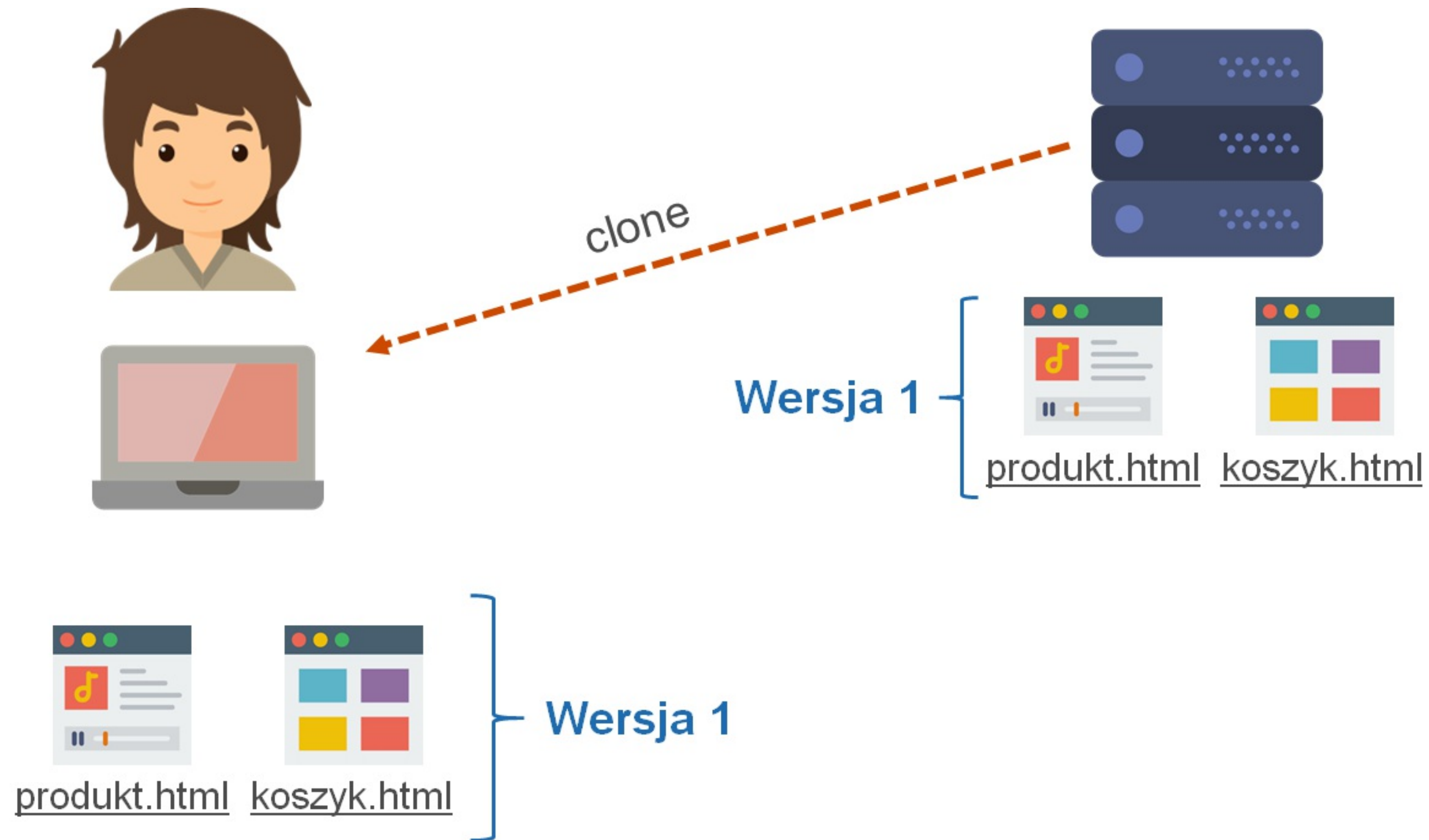
# Praca zespołowa

## Marcin dołącza do projektu

3. Marcin musi odejść od komputera. Gdy wraca, chce sprawdzić, co już zmienił:

```
marcin@xwing:~/workspace/sklep$ 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)
    zmodyfikowany: produkt.html
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    zamowienie.html
no changes added to commit (use "git add" and/or "git commit -a")
marcin@xwing:~/workspace/sklep$
```

# Praca zespołowa



# Praca zespołowa

## Marcin dołącza do projektu

4. Marcin dodaje zmiany do Gita:

- **produkt.html**
- **zamowienie.html**

```
git add .  
git commit -m "produkt i zamówienie"
```





# Praca zespołowa

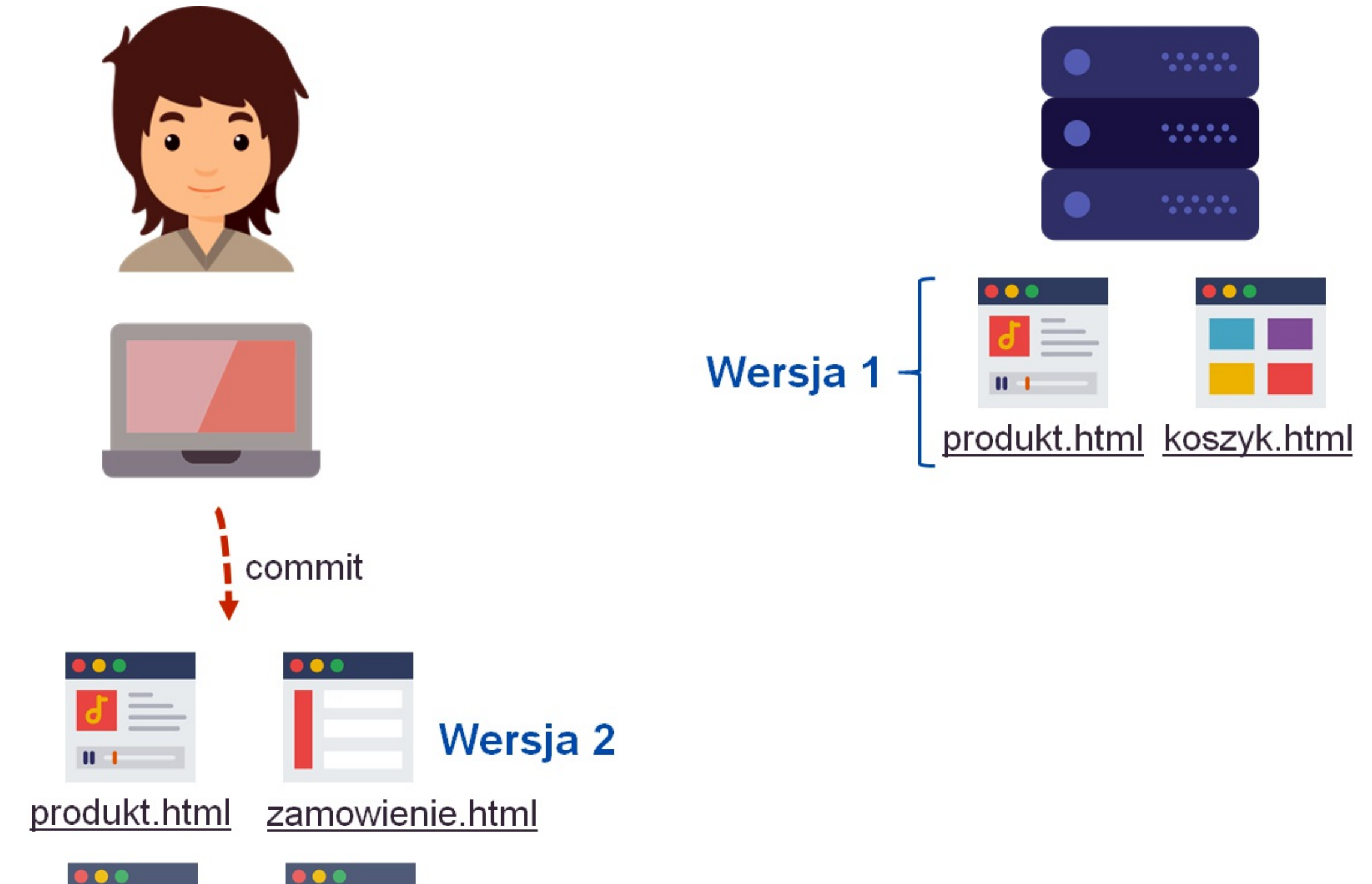
## Marcin dołącza do projektu

4. Marcin dodaje zmiany do Gita:

- **produkt.html**
- **zamowienie.html**

```
git add .  
git commit -m "produkt i zamówienie"
```

**Uwaga:** ta kropka jest tu nieprzypadkowo!  
Oznacza ona wszystkie zmienione lub  
dodane pliki z aktualnego folderu i  
wszystkich folderów poniżej.

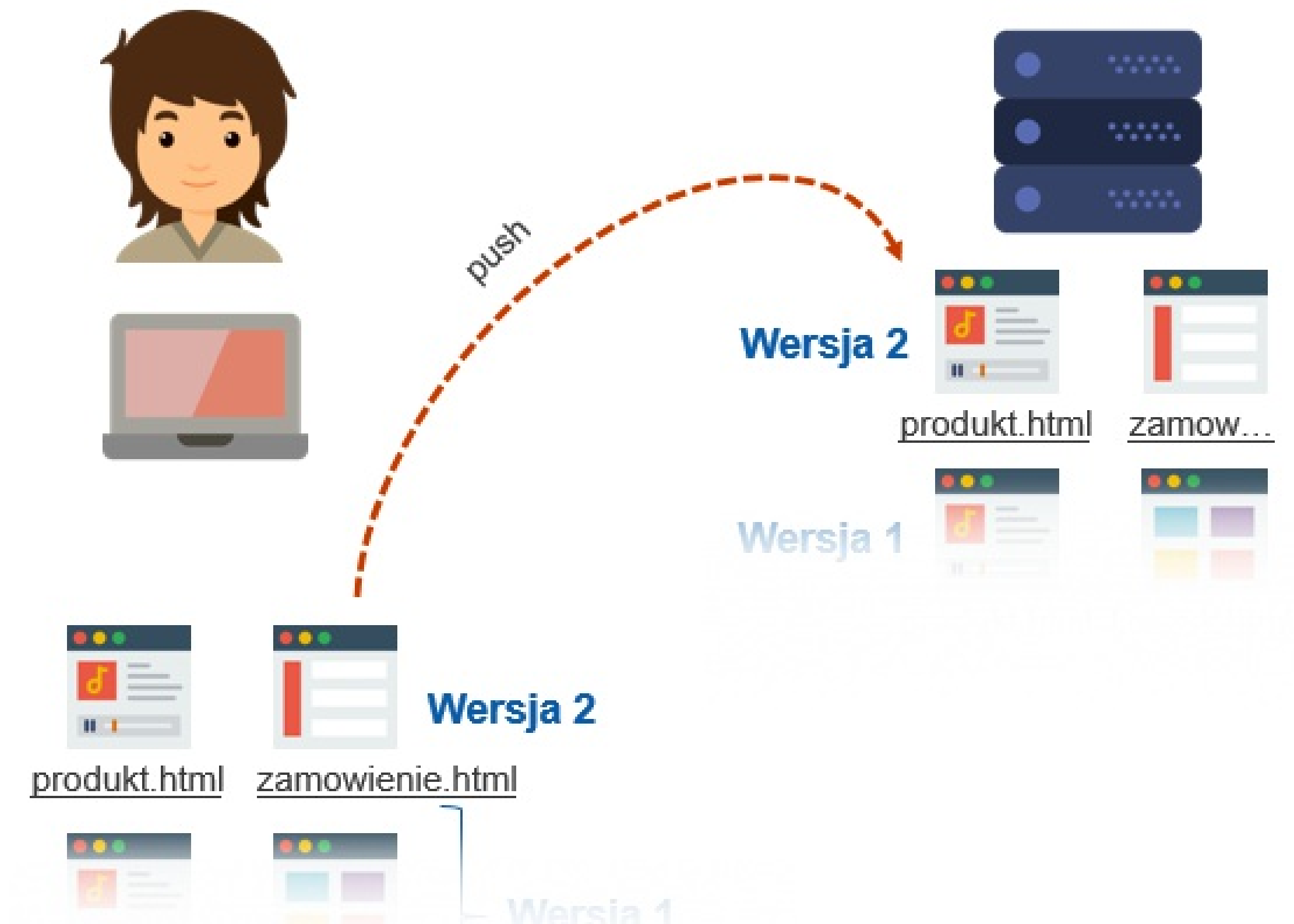


# Praca zespołowa

## Marcin dołącza do projektu

5. Marcin wypycha (**push**) swoje zmiany do **repozytorium zdalnego**.

```
git push
```



# Praca zespołowa

## Co się stanie, gdy Agata wróci do pracy nad zdalnym repozytorium?

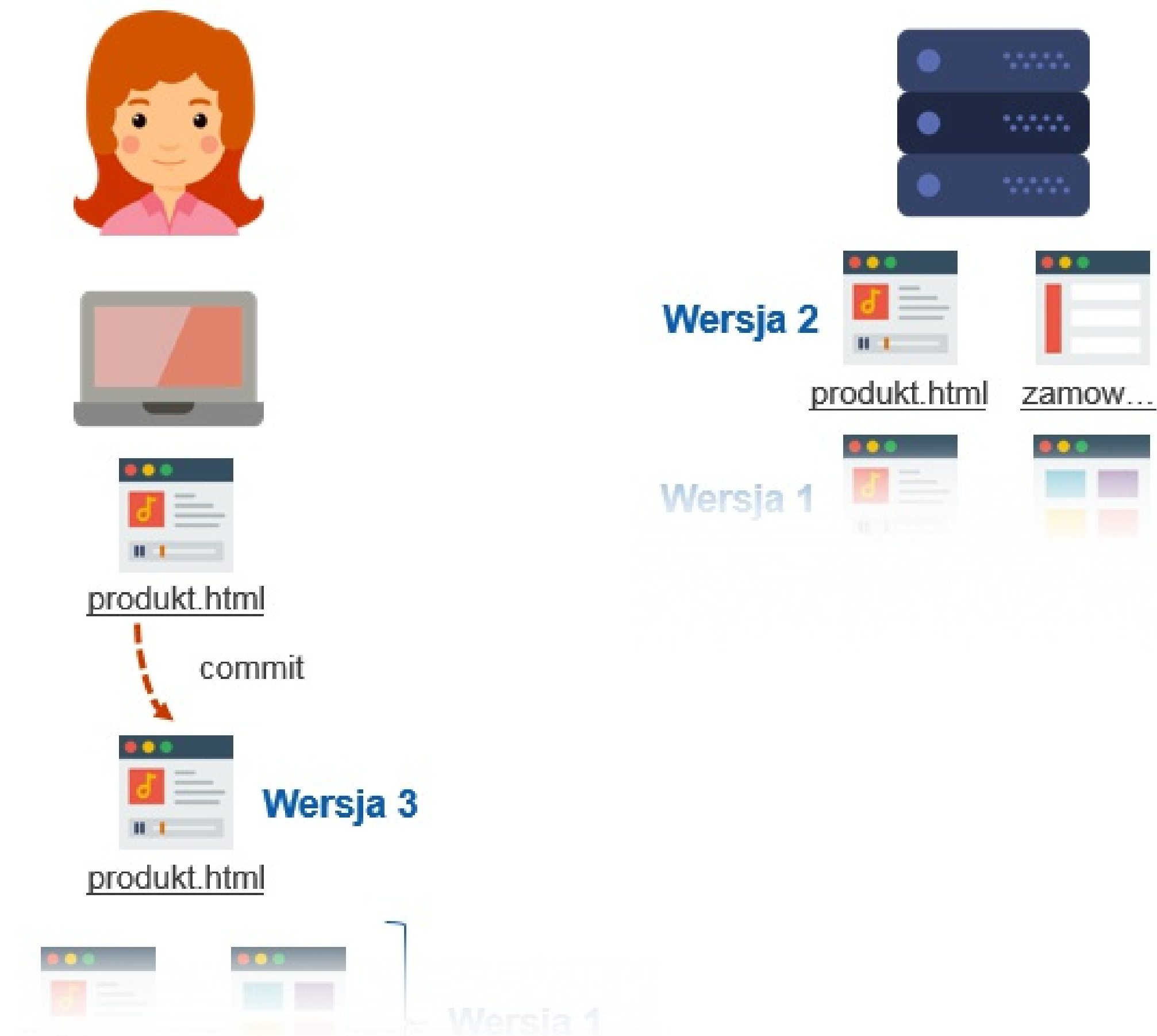
6. Agata ma u siebie **wersję 1**, Marcin dodał do zdalnego repozytorium **wersję 2**.

Agata zmienia jeden plik:

➤ **produkt.html**

Agata dodaje plik do kontroli wersji i zatwierdza zmiany:

```
git add produkt.html  
git commit -m "praca nad produktem"
```



# Praca zespołowa

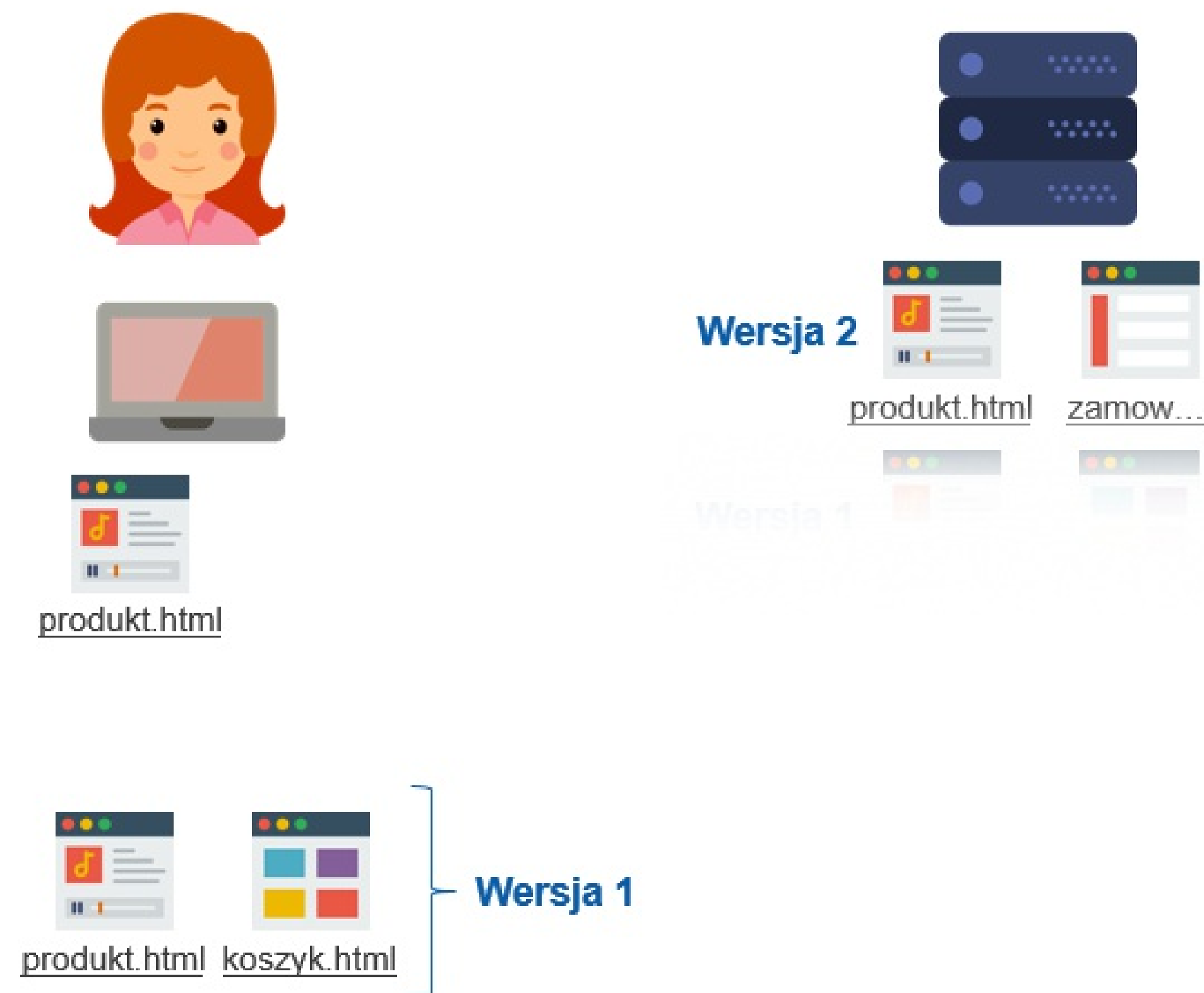
## WTEM!

7. Gdy Agata próbuje wypchnąć plik (**git push**), dostaje poniższy komunikat. Git poinformował Agatę, że w zdalnym repozytorium znajdują się zmiany zrobione przez Marcina. Uniemożliwiają one wrzucenie zmian Agaty.

## Czy problem Agaty da się rozwiązać?

```
! [rejected] master -> master (fetch first)
error: failed to push some refs to
"https://github.com/marcin-barylka/sklep-internetowy.git"
HINT: Updates were rejected because the remote contains work that you do
HINT: not have locally. This is usually caused by another repository pushing
HINT: to the same ref. You may want to first integrate the remote changes
HINT: (e.g., "git pull ...") before pushing again.
HINT: See the "Note about fast-forwards" in "git push --help" for details.
```

# Praca zespołowa



# Praca zespołowa

## Nic złego się nie stało

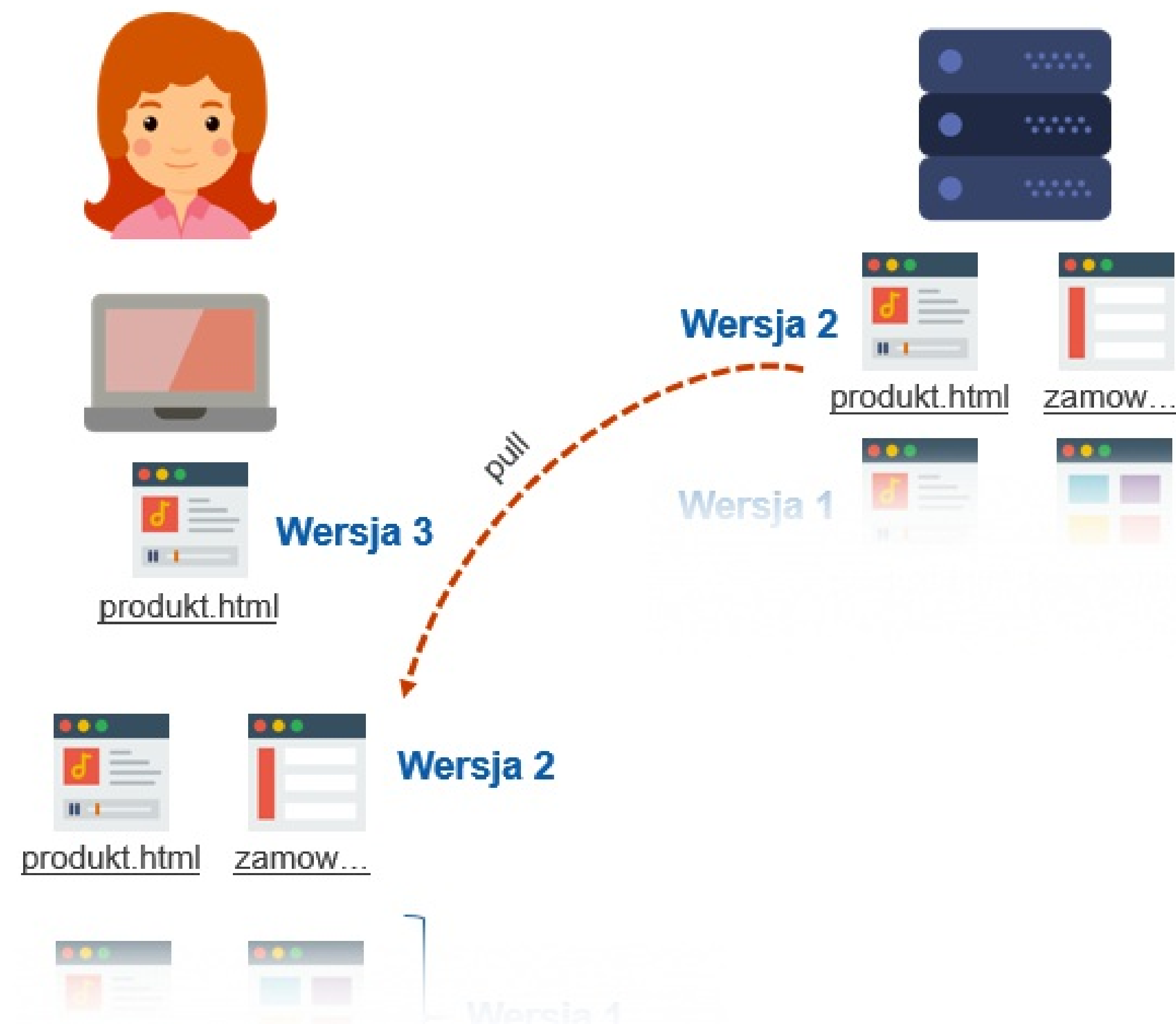
8. Agata musi tylko zintegrować zmiany Marcina ze swoimi. Powinna zatem wykonać instrukcję **pull**.

```
git pull
```

**Git proponuje Agacie opis zmiany. Wystarczy go zaakceptować.**

```
Merge branch "master" of https://github.com/marcin-barylka/sklep-internetowy
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with "#" will be ignored, and an empty message aborts the
commit.
```

# Praca zespołowa





# Konflikt!

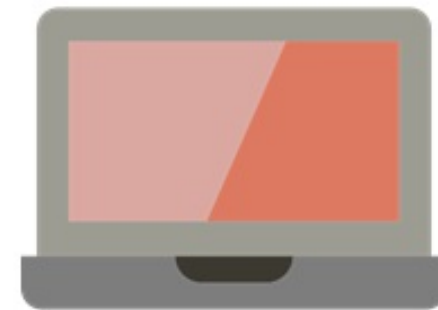


# Konflikt!

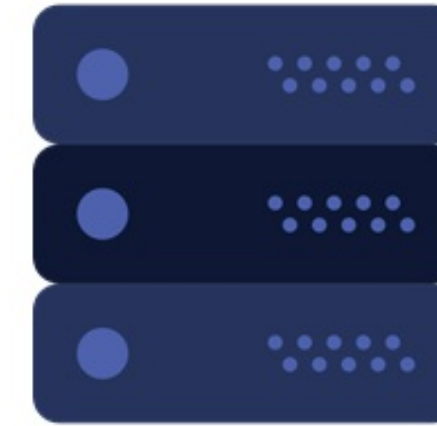
Agata pracuje nad plikiem **zamówienie.html**. Umieszcza plik w zdalnym repozytorium. Kilka dni później Marcin też pracuje nad tym samym plikiem. Również próbuje umieścić go w zdalnym repozytorium. Dostaje jednak znany nam komunikat o czekających zmianach w repozytorium zdalnym. Marcin próbuje ściągnąć te zmiany komendą **git pull**.

```
! [rejected]master -> master (fetch first)
error: failed to push some refs to
"https://github.com/marcin-barylka/sklep-internetowy.git"
HINT: Updates were rejected because the remote contains work that you do
HINT: not have locally. This is usually caused by another repository pushing
HINT: to the same ref. You may want to first integrate the remote changes
HINT: (e.g., "git pull ...") before pushing again.
HINT: See the "Note about fast-forwards" to "git push --help" for details.
agatalagata:~/workspace/sklep-internetowy$
```

# Konflikt!



zamowienie.html



zamow...

# Konflikt!

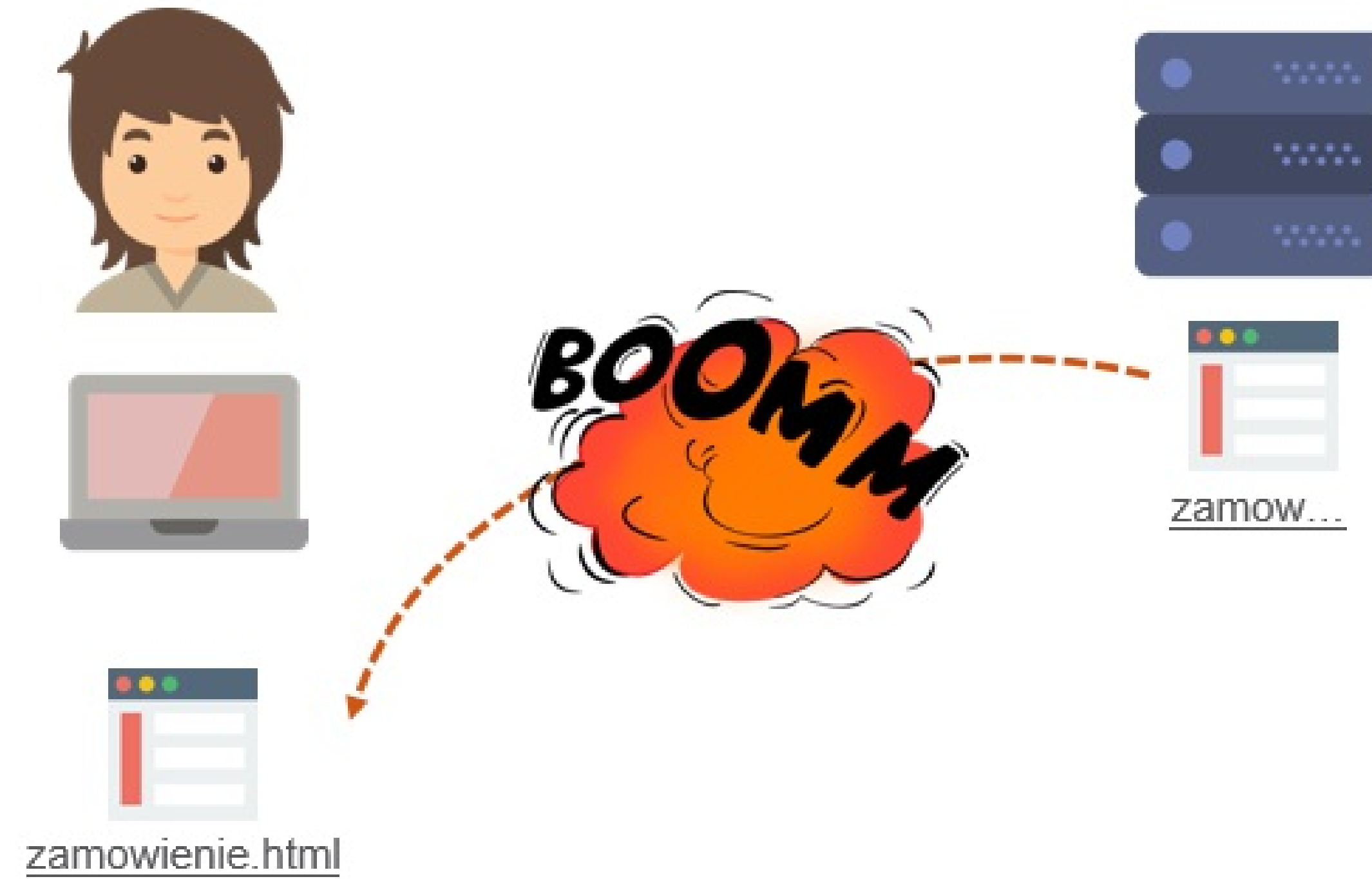
## Wtem!

```
marcin@xwing:~/workspace/sklep-internetowy$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/marcin-barylka/sklep-internetowy
   c9d6e9d..38d8078 master -> origin/master
Auto-merging zamowienie.html
CONFLICT (zawartość): Merge conflict in zamowienie.html
Automatic merge failed; fix conflicts and then commit the result.
marcin@xwing:~/workspace/sklep-internetowy$
```

## Git informuje Marcina o konflikcie!

Czy oznacza to kłótnię Marcina z Agatą?

# Konflikt!



# Konflikt!

## Konflikt w kodzie to nic złego

Git nie potrafił połączyć automatycznie zmian Marcina i Agaty.

Marcin musi połączyć te zmiany ręcznie,  
a następnie wykonać **commit** i **push**.

Wbrew pozorom to duże ułatwienie dla programistów. Mimo wszystko mogą pracować równocześnie na tym samym pliku.

Git automatycznie przypilnuje, aby nie psuli sobie nawzajem kodu. Jeśli nie poradzi sobie z integracją zmian, poprosi o to programistę.

```
<<<<<< HEAD
=====
>>>>>> 38d80788cceb005ed48aa225cb5668d8704474b
        <h1>Zamówienie</h1>
```

# Konflikt!

Co oznaczają nowe linijki które pojawiły się w pliku?

```
<<<<<< HEAD
=====
>>>>>> 38d80788ccebcb005ed48aa225cb5668d8704474b
        <h1>Zamówienie</h1>
```

# Konflikt!

Co oznaczają nowe linijki które pojawiły się w pliku?

**HEAD** to najświeższe zmiany z **lokalnego repozytorium** (czyli te, które zrobił przed chwilą Marcin). Zaczynają się od tej linijki.

```
<<<<<< HEAD
=====
>>>>>> 38d80788ccebcb005ed48aa225cb5668d8704474b
        <h1>Zamówienie</h1>
```



# Konflikt!

Co oznaczają nowe linijki które pojawiły się w pliku?

**HEAD** to najświeższe zmiany z **lokalnego repozytorium** (czyli te, które zrobił przed chwilą Marcin). Zaczynają się od tej linijki.

Ta linia pokazuje koniec zmian z **lokalnego repozytorium**. Poniżej niej możemy znaleźć zmiany z **zdalnego repozytorium**, czyli zmiany Agaty sprzed kilku dni.

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

```
=====
```

```
>>>>>> 38d80788ccebcb005ed48aa225cb5668d8704474b
```

```
<h1>Zamówienie</h1>
```

# Konflikt!

Co oznaczają nowe linijki które pojawiły się w pliku?

**HEAD** to najświeższe zmiany z **lokalnego repozytorium** (czyli te, które zrobił przed chwilą Marcin). Zaczynają się od tej linijki.

Ta linia pokazuje koniec zmian z **lokalnego repozytorium**. Poniżej niej możemy znaleźć zmiany z **zdalnego repozytorium**, czyli zmiany Agaty sprzed kilku dni.

Ta linijka oznacza koniec zmian z **zdalnego repozytorium**. Długa liczba, zaczynająca się od **38d807**, to identyfikator najnowszego commitu znajdującego się na **zdalnym repozytorium**.

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

```
=====
```

```
>>>>>> 38d80788ccebcb005ed48aa225cb5668d8704474b
```

```
<h1>Zamówienie</h1>
```

# Konflikt!

## Marcin rozwiązuje konflikt.

Polega to na wykonaniu następujących kroków:

1. Usunięciu linijek automatycznie dodanych przez git (czyli linijki 1, 5 i 7).
2. Łączy kod tak żeby zachować zmiany zarówno swoje jak i Agaty. Wymaga to zrozumienia tych zmian i ręcznego ich połączenia.

## Poprawiony plik:

```
<h1>Zamówienie</h1>
```

# Konflikt!

## Marcin rozwiązuje konflikt

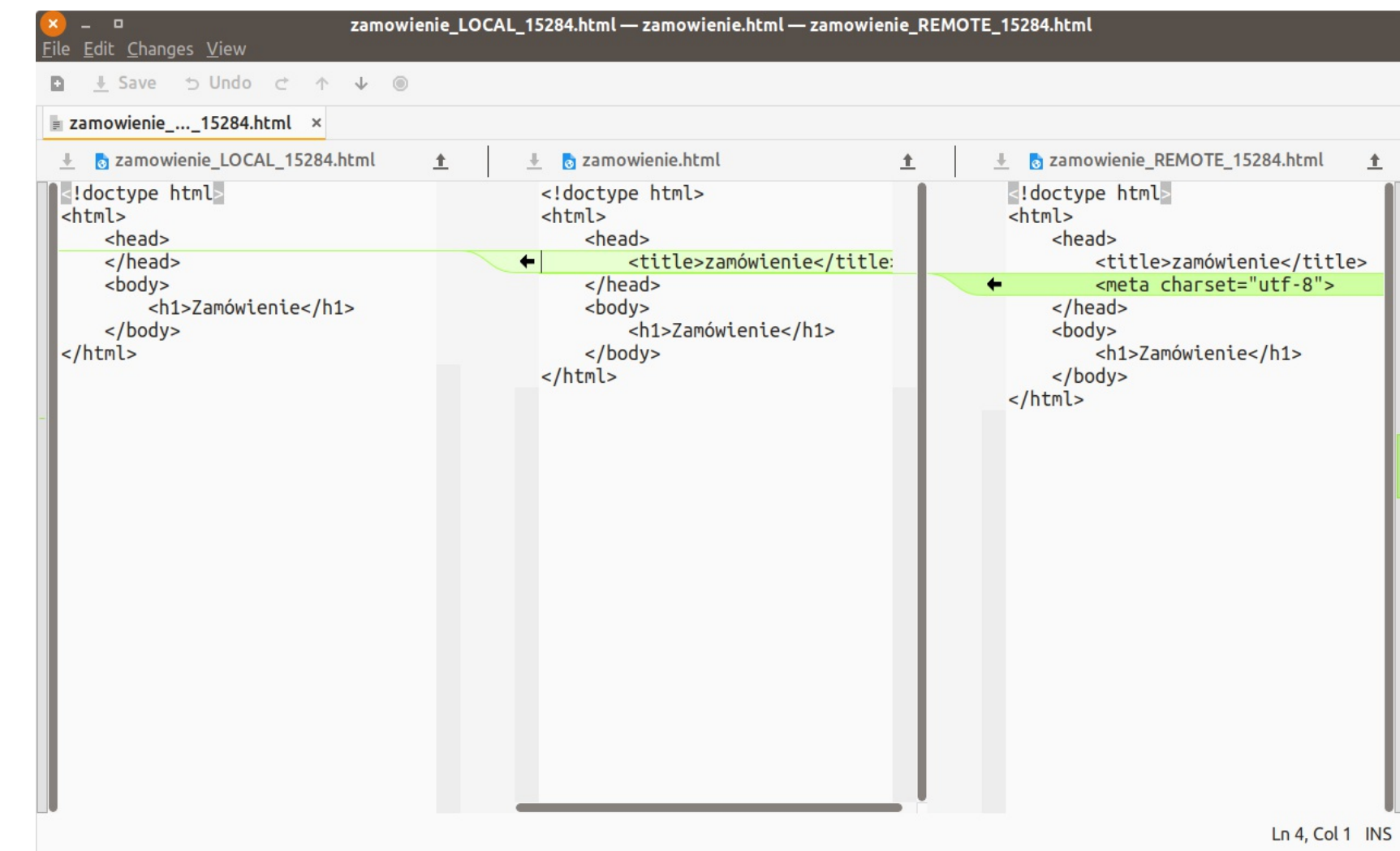
3. Dodaje zmiany do kontroli wersji (**git add**).
4. Wykonuje **git commit** z komentarzem o rozwiązaniu konfliktu.
5. Wypycha zmiany do repozytorium zewnętrznego.

```
marcin@xwing:~/workspace/sklep-internetowy$ git add zamowienie.html
marcin@xwing:~/workspace/sklep-internetowy$ git commit -m "Rozwiązanie konfliktu"
[master b1de440] Rozwiązanie konfliktu
marcin@xwing:~/workspace/sklep-internetowy$ git push
```

# Narzędzie do usuwania konfliktów

# Narzędzie do łączenia kodu

Ręczne usuwanie konfliktów jest czasochłonne, żmudne i może powodować błędy. Aby ułatwić sobie nieco życie można użyć odpowiedniego narzędzia. Takim narzędziem jest np. **meld**.



# meld

## Instalacja

Jeśli używasz naszego skryptu instalacyjnego program **meld** masz już zainstalowany i skonfigurowany. Jeśli nie, to na **Ubuntu** możesz zrobić to komendą:

```
sudo apt install meld
```

Na **MacOS** zrobisz to następująco:

```
brew cask install meld
```

Konfiguracja odbywa się następująco:

```
git config --global merge.tool meld
```

Powyższa linijka oznacza, że standardowym narzędziem do rozwiązywania konfliktów w kodzie dla Gita, będzie od tej pory **meld**.



# Narzędzie do integrowania zmian

## Rozwiązywanie konfliktów

Teraz możemy rozwiązywać konflikty nieco łatwiej. Wystarczy wpisać w terminalu komendę:

```
git mergetool
```

Git teraz przejrzy wszystkie konflikty i zapyta Cię co robić:

```
$ git mergetool  
Merging:  
zamowienie.html
```

```
Normal merge conflict for 'zamowienie.html':  
  {local}: modified file  
  {remote}: modified file
```

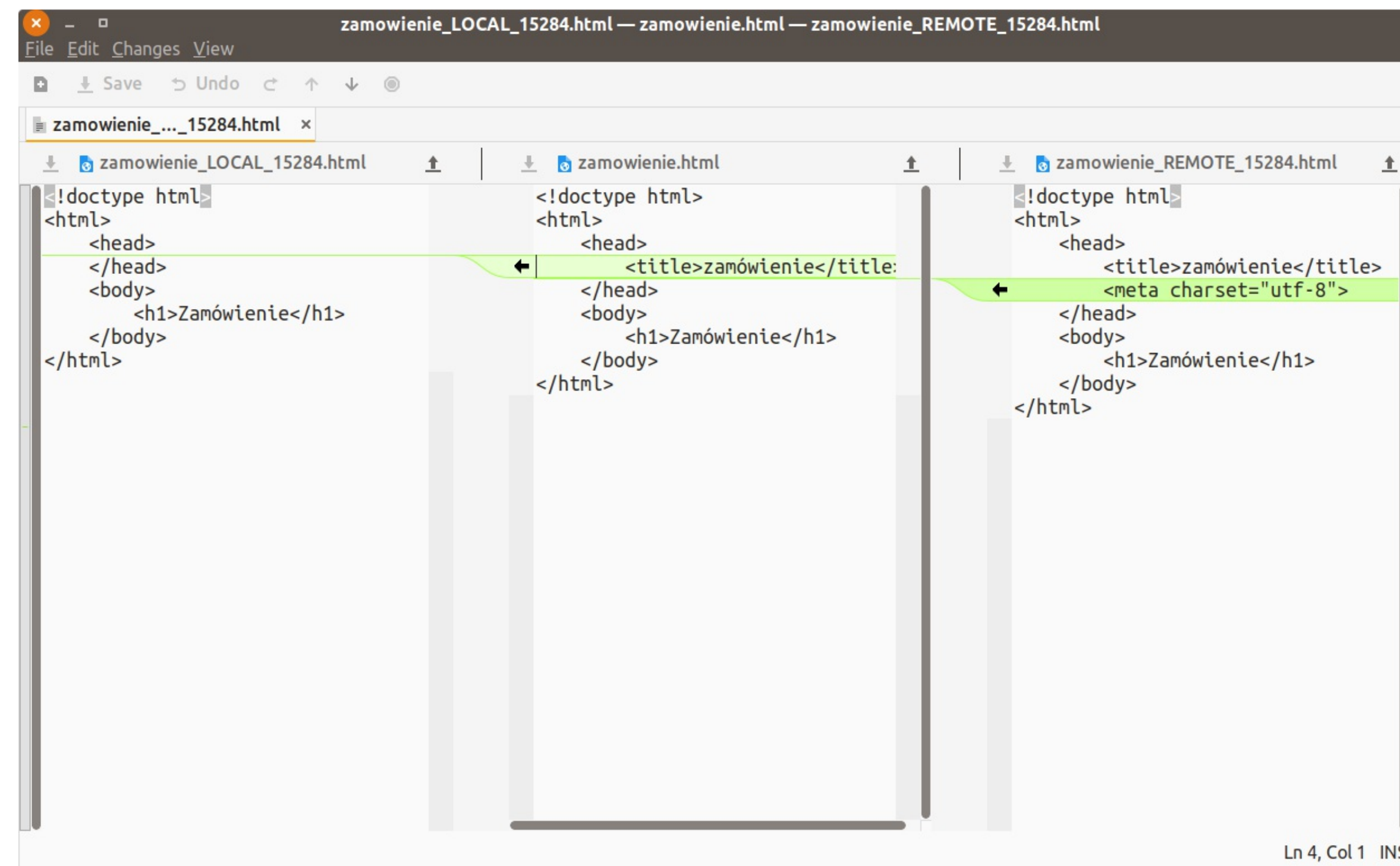
# meld

Zobaczmy okno z trzema panelami:

```
zamowienie_LOCAL_15284.html | zamowienie.html | zamowienie_REMOTE_15284.html
File Edit Changes View
+ Save Undo C ↑ ↓
zamowienie_..._15284.html x
zamowienie_LOCAL_15284.html zamowienie.html zamowienie_REMOTE_15284.html
<!doctype html> <!doctype html> <!doctype html>
<html> <html> <html>
<head> <head> <head>
</head> <title>zamówienie</title: <title>zamówienie</title>
<body> </head> <meta charset="utf-8">
<h1>Zamówienie</h1> <body> </head>
</body> <h1>Zamówienie</h1> <body>
</html> </body> <h1>Zamówienie</h1>
</html> </html> </body>
Ln 4, Col 1 INS
```

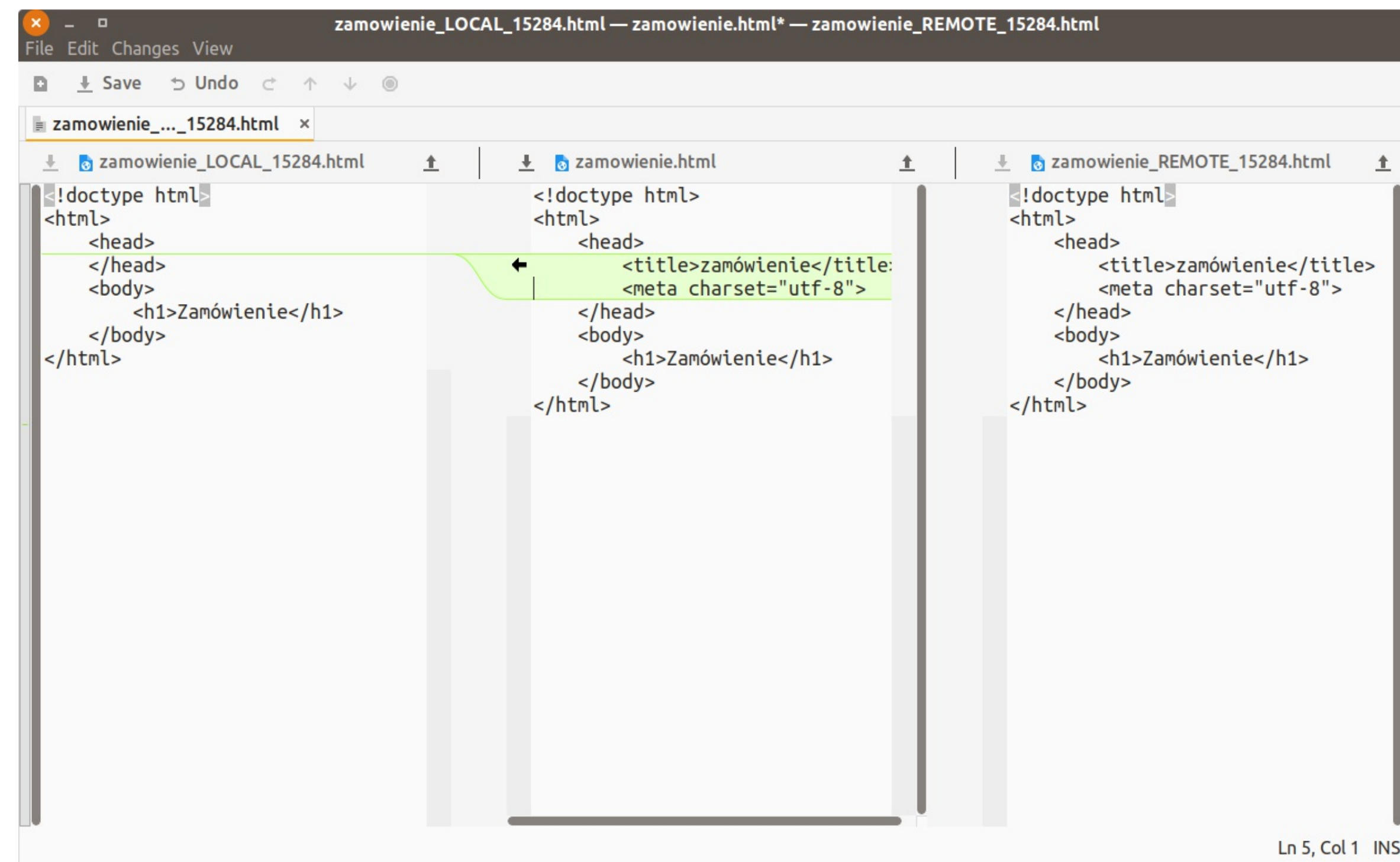
# meld

- Lewy panel narzędzia to **lokalne** zmiany czyli branch, na którym aktualnie się znajdujemy.
- Prawy panel – repozytorium **zdalne** lub **branch, który łączymy z aktualnym**.
- Środkowy panel – plik wynikowy.



# meld

Klikając odpowiednie strzałki, możemy zintegrować zmiany w kodzie do pliku wynikowego. Np. klikając w strzałkę w prawo (prawy panel) przeniesiemy element `<meta>` do środkowego panelu:



# Integracja zmian

Po zapisaniu zmian (CTRL-s) i rozwiązaniu wszystkich konfliktów, należy jeszcze zrobić **commit** zintegrowanych zmian:

```
git add .  
git commit -m "rozwiązanie konfliktu"
```

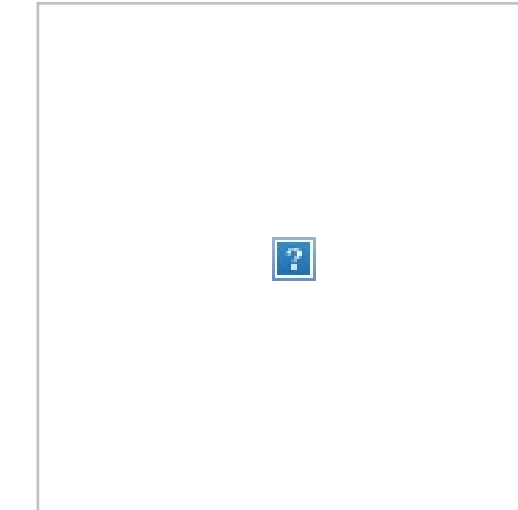
...i gotowe!

# **Branches, czyli odgałęzienia kodu**



# Branch

**Marcin**, w trakcie tworzenia projektu, wpadł na pomysł, że może trochę inaczej rozmieścić elementy na stronie. Jego kierownik stwierdził, że może to być interesujący pomysł i polecił mu poeksperymentować z kodem źródłowym. **Marcin** nie chce jednak psuć **Agacie** jej pracy – gdyby eksperymenty się nie udały, byłby kłopot. Jednak, gdyby jego próby okazały się sukcesem – należałoby wdrożyć tę zmianę.





# Branch

## Marcin tworzy odgałęzienie (branch) kodu

Odgałęzienie (ang. *branch*), to kopia kodu w repozytorium, którą można rozwijać niezależnie od głównej wersji. Używa się tego do eksperymentowania na kodzie albo (częściej) do dodawania nowych funkcji w taki sposób, aby nie zepsuć gotowej i działającej wersji.

### Tworzenie brancha:

Aby utworzyć nowy branch należy wpisać w terminalu:

```
git checkout -b <nazwa-brancha>
```

Czyli, jeśli **Marcin** chce utworzyć branch o nazwie *eksperyment*, musi napisać:

```
git checkout -b eksperyment
```

# Branch

## Przełączanie między gałęziami

Gdy **Marcin** wpisze w terminalu:

```
git branch
```

może zobaczyć jakie gałęzie są utworzone i, na której aktualnie się znajduje.

```
$ git branch  
* eksperyment  
master
```

**master**, to główna gałąź kodu. Ten branch jest tworzony automatycznie, gdy zakładamy repozytorium. Branch **eksperyment**, to gałąź założona przez **Marcina**. Obok nazwy brancha widnieje gwiazdka: oznacza to, że **Marcin** aktualnie znajduje się na tej gałęzi kodu.

# Branch

W każdej chwili można przełączyć się między gałęziami. Wystarczy wydać komendę:

```
git checkout <nazwa-gałęzi>
```

Jednak, zanim **Marcin** się przełączy, musi zrobić **commit** swoich zmian:

```
git add zamowienie.html  
git commit -m "eksperymentalne zmiany"  
git checkout master
```

# Branch

## Łączenie gałęzi

**Marcin** skończył pracę nad eksperymentalną gałęzią kodu i uzyskał zgodę na połączenie jej z główną linią oprogramowania. Musi zatem połączyć (**merge**) swoje zmiany z gałęzią główną:

Najpierw musi przełączyć się na gałąź, na której znajduje się główna wersja kodu:

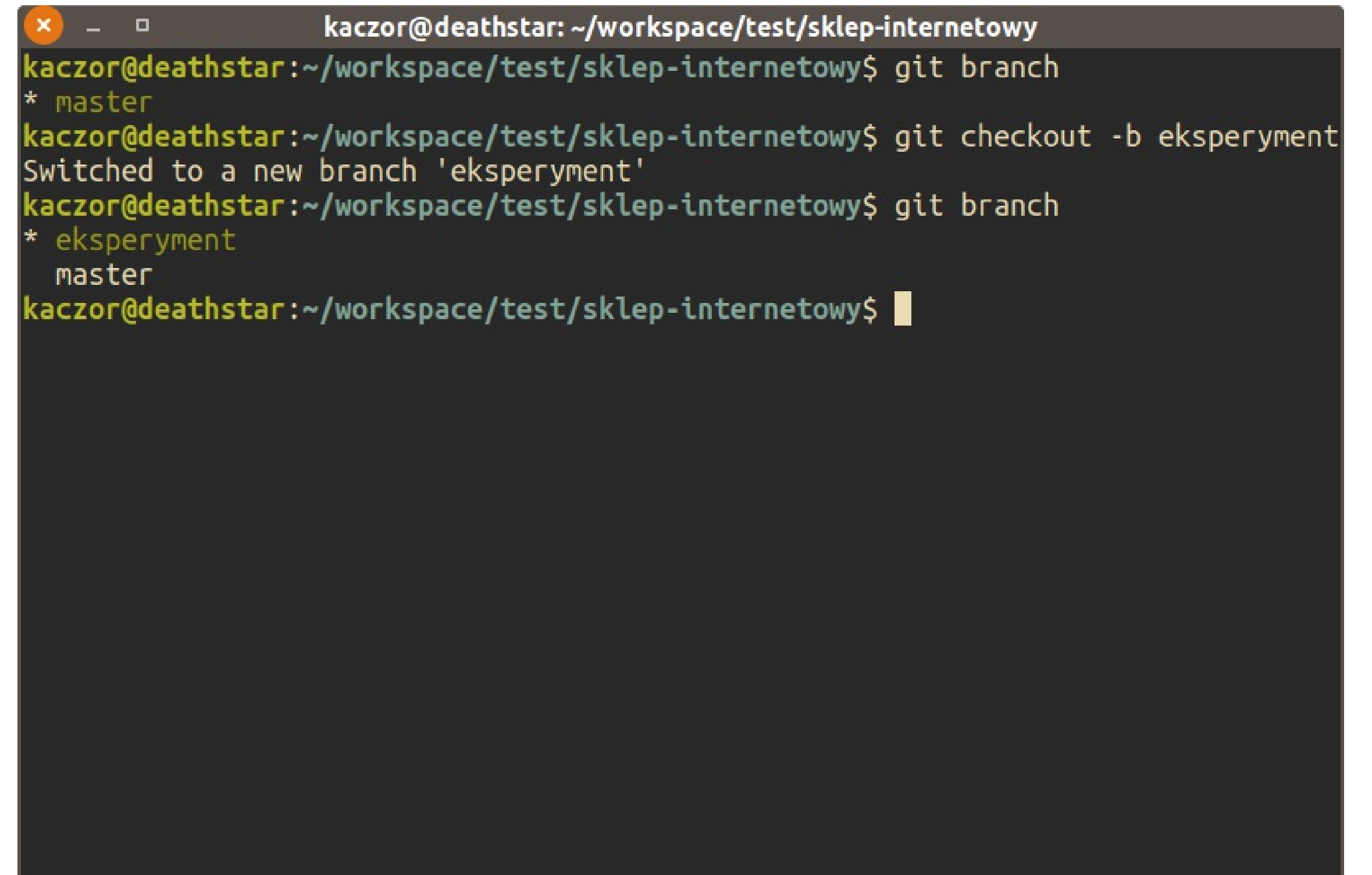
```
git checkout master
```

Następnie wydać komendę **merge**:

```
git merge <nazwa-gałęzi-ze-zmianami>
```

czyli w przypadku eksperymentu **Marcina**:

```
git merge eksperyment
```

A terminal window with a dark background and light-colored text. The window title is 'kaczor@deathstar: ~/workspace/test/sklep-internetowy'. The terminal shows the following commands and output: 1. 'git branch' returns '\* master'. 2. 'git checkout -b eksperyment' returns 'Switched to a new branch 'eksperyment''. 3. 'git branch' returns '\* eksperyment' and 'master'. The cursor is at the end of the last command line.

```
kaczor@deathstar: ~/workspace/test/sklep-internetowy
kaczor@deathstar:~/workspace/test/sklep-internetowy$ git branch
* master
kaczor@deathstar:~/workspace/test/sklep-internetowy$ git checkout -b eksperyment
Switched to a new branch 'eksperyment'
kaczor@deathstar:~/workspace/test/sklep-internetowy$ git branch
* eksperyment
  master
kaczor@deathstar:~/workspace/test/sklep-internetowy$
```

**Uwaga:** po połączeniu gałęzi mogą wystąpić konflikty. Należy je rozwiązać używając odpowiednich narzędzi.

**Plik  
.gitignore**

# .gitignore

Niektóre pliki w naszym projekcie są przeznaczone tylko dla nas i nie chcemy ich umieszczać w repozytorium. Są to, na przykład:

- pliki konfiguracyjne Twojego edytora programistycznego (bo inni programiści mogą używać innego edytora i te pliki nie są im potrzebne),
- pliki z hasłami do bazy danych (ze względów bezpieczeństwa),
- skompilowane pliki binarne (np. pliki **.class**, **.pyc** albo podobne),
- ...i wiele innych.

**Jak zapobiec umieszczaniu takich plików w repozytorium?**

# .gitignore

## Źle! :(

Za każdym razem, gdy dodajesz pliki do kontroli wersji, wpisujesz ich nazwy, pamiętając o tym, by nie dodać niepotrzebnych:

```
$ git add klient.html katalog.html
```

...bo możesz się pomylić i dodać za dużo. Pamiętaj: nawet jeśli później usuniesz te pliki, to stare wersje zostają w repozytorium już na zawsze!

## Dobrze :)

Należy utworzyć plik **.gitignore**, gdzie umieścimy listę plików i katalogów, które nie zostaną dodane do repozytorium.

**Uwaga:** pamiętaj o tym, że pliki zaczynające się kropką, to w Linuksach i MacOS pliki ukryte!



# .gitignore

## Budowa pliku .gitignore

Wewnątrz pliku **.gitignore**, w kolejnych liniach wpisujesz nazwy plików lub katalogów, np:

```
notatki.txt  
pycache/
```

Do repozytorium **nie** zostaną dodane:

# .gitignore

## Budowa pliku .gitignore

Wewnątrz pliku **.gitignore**, w kolejnych liniach wpisujesz nazwy plików lub katalogów, np:

```
notatki.txt  
pycache/
```

Do repozytorium **nie** zostaną dodane:

→ plik **notatki.txt**,

# .gitignore

## Budowa pliku .gitignore

Wewnątrz pliku **.gitignore**, w kolejnych liniach wpisujesz nazwy plików lub katalogów, np:

```
notatki.txt  
pycache/
```

Do repozytorium **nie** zostaną dodane:

- plik **notatki.txt**,
- katalog **pycache/** i wszystkie pliki w nim zawarte.

# .gitignore

## Budowa pliku .gitignore - ciąg dalszy

Możesz również używać gwiazdek: pojedyncza oznacza dowolny ciąg znaków w nazwie pliku, np.

```
*.class
```

oznacza, że do repozytorium **nie** zostaną dodane pliki, których nazwa kończy się znakami **.class**.

**Dwie** gwiazdki zastępują nazwę dowolnego katalogu, np:

```
**/logs
```

oznaczają, że do repozytorium nie zostaną dodane następujące pliki:

- **logs/debug.log,**
- **logs/2018/06/04/access.log,**
- **build/logs/debug.log.**

# .gitignore

## Budowa pliku .gitignore - ciąg dalszy

```
logs/**/debug.log
```

oznacza, że do repozytorium nie zostaną dodane następujące pliki:

- `logs/debug.log`,
- `logs/2018/06/04/debug.log`,
- `logs/build/debug.log`,

ale pliki:

- `logs/access.log`,
- `logs/2018/06/04/build.log`,
- `logs/build/output.txt`,

będą w repozytorium.

# .gitignore

Dobry opis wewnętrznej składni pliku **.gitignore** znajdziesz tutaj:

- <https://www.atlassian.com/git/tutorials/saving-changes/gitignore>
- <https://git-scm.com/docs/gitignore>

## Skomplikowane?

Bez paniki! na stronie <https://www.gitignore.io/> znajdziesz kreator plików **.gitignore**!

# Problemy z Gitem



# Github nie pozwala zrobić operacji push

Jeżeli nie możesz wypchnąć (**git push**) swoich zmian, prawdopodobnie sklonowałeś oryginalne repozytorium, a nie swojego forka. Komunikat w terminalu jaki się pojawia przy tego typu problemie to:

```
kaczor@deathstar:~/workspace/spinning-cube$ git push origin master  
ERROR: Permission to Kaczor2704/spinning-cube.git denied to marcin-barylka.  
fatal: Could not read from remote repository.
```

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

# Github nie pozwala zrobić operacji push

## Jak naprawić taki błąd?

- Wejdź na swoje konto Github i znajdź fork repozytorium, z którym masz problem.
- Wybierz opcję **Clone or download** i skopiuj adres repozytorium.

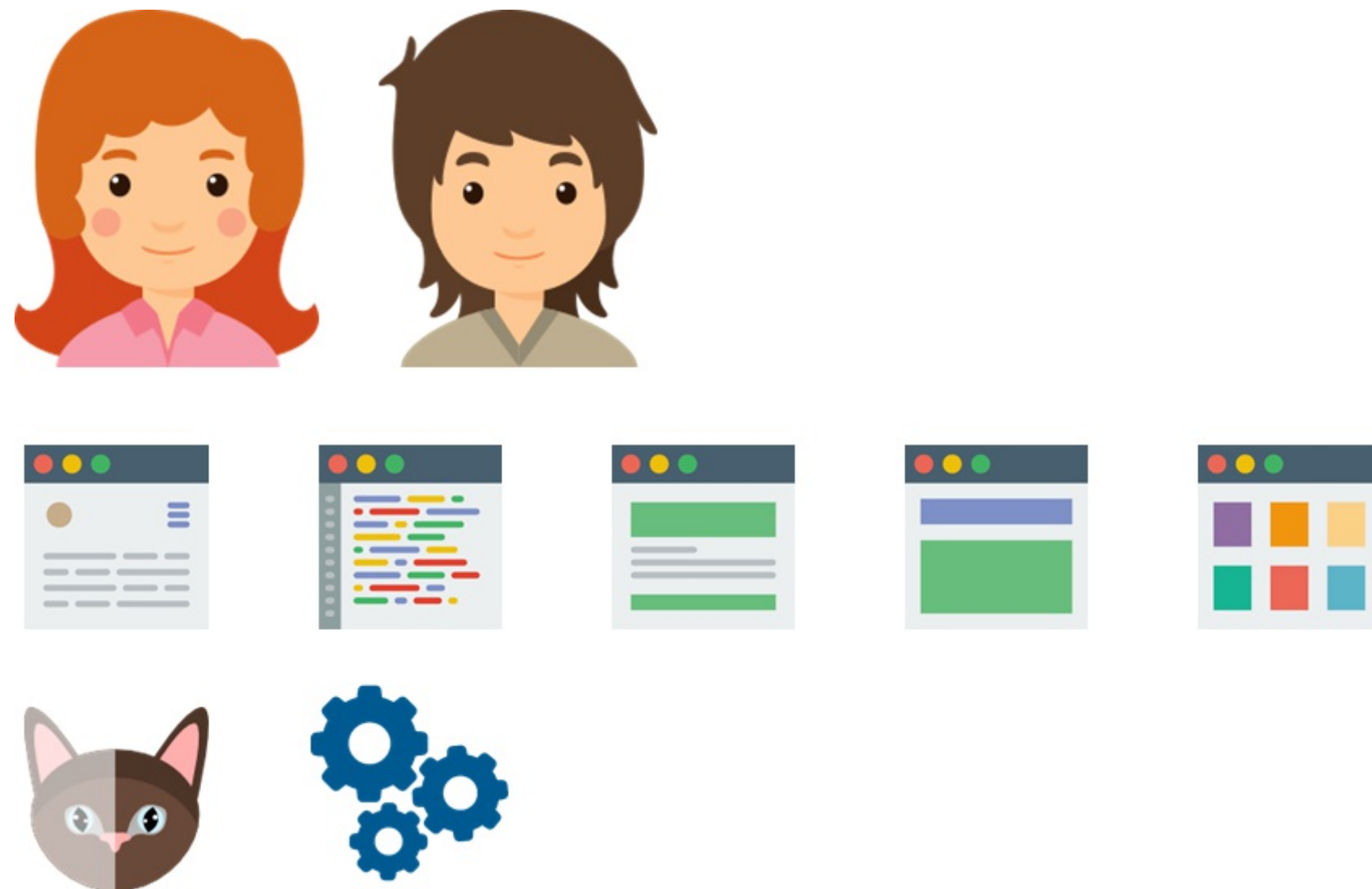
Adres powinien mieć taką strukturę: `https://github.com/<twój_login>/<nazwa_repozytorium.git>`.

- Otwórz terminal i przejdź do katalogu z problematycznym repozytorium.
- Wpisz komendę:  
`git remote set-url origin`  
`https://github.com/<twój_login>/<nazwa_repozytorium.git>`  
i wciśnij Enter.

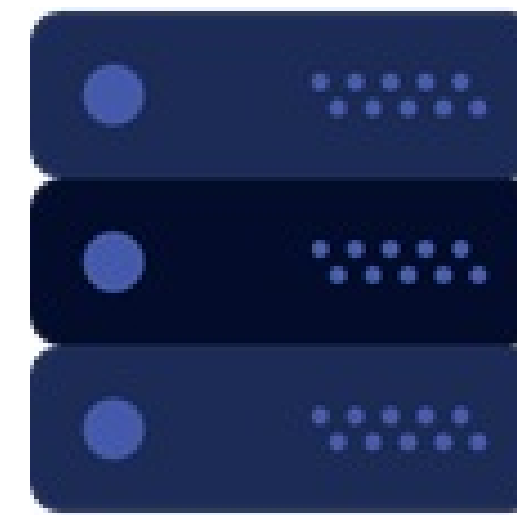
# **Obowiązkowa informacja o prawach autorskich**

# Copyrights

Icons designed by **Freepik** from  
**www.flatcon.com**



Icons designed by **Madebyolivier** from  
**www.flatcon.com**



# **Przydatne linki dla użytkowników Gita**

# Przydatne linki

- Strona główna projektu Git  
<https://git-scm.com>
- Oficjalny podręcznik do Gita (za darmo, w różnych formatach)  
<https://git-scm.com/book/en/v2>
- Ciekawy samouczek Gita, na stronie GitHuba  
<https://try.github.io/levels/1/challenges/1>
- Kolejny tutorial do Gita  
<https://www.git-tower.com/learn/git/ebook/en/mac/introduction>



# Git – zadanie

- Spróbuj przejrzeć linki dostępne poniżej i wybierz przynajmniej dwa, które przerobisz obowiązkowo.
  - <http://gitreal.codeschool.com/levels/1>
  - <http://www.git-tower.com/learn/git/videos>
  - <http://www.dataschool.io/git-and-github-videos-for-beginners>
- Dla bardziej zaawansowanych
  - <http://pcottle.github.io/learnGitBranching>