**Git And Github**

**الفرق بين clone , download**

**ال clone : بتاخد انت نسخه وبيكون لسه في relation بينك وبين السيرفر او المشروع الي انت نزلته يعني تقدر تعمل تعديلات معينة علي الشغل بتاعك بعد كدا تعملها push الي هو ترفع الشغل بتاعك والشخص ال author المسئول عن المشروع يقرر بقا هو هيوافق علي الشخبطة الي انت عملتها في الكود دي ولا يرفض , وكمان لو مثلا الشخص المسئول دا عمل تعديلات اضافية بعد ما انت عملت حاجة اسمها clone دي انت كمان تقدر تسحب التعديلات الي هو عملها دي عن طريق حاجة اسمها pull وهكذا يعني .  
download : انت خلاص نزلت المشروع عندك وفقدت الاتصال تماما بالسيرفر او اي حاجة تخص المشروع دا .**

**المميزات**

**ما تقوله هو خصائص وميزات Git كنظام إدارة النسخ الموزعة**

**(Distributed Version Control System DVCS).**

**دعني أوضح لك كل من هذه الخصائص:**

**1. Distributed (موزعة): في Git، كل نسخة من المشروع تحتوي على كامل تاريخ التغييرات (سجل الاصدارات)، لذلك يمكن لكل مستخدم العمل بشكل مستقل وإجراء التغييرات دون الحاجة للاتصال بالخادم الرئيسي. هذا يعزز التعاون ويجعل عملية التطوير أكثر فعالية.**

**2. Speed (السرعة): Git مصمم ليكون سريعًا، سواء كان في عمليات النسخ (clone) أو الانتقال بين الفروع (branches) أو الإرجاع إلى الإصدارات السابقة. هذا يجعله مناسبًا للمشاريع الكبيرة والصغيرة على حد سواء.**

**3. Simple Design (تصميم بسيط): Git مصمم بتصميم بسيط وفعال، مما يجعله سهل الاستخدام والفهم، ويسمح للمستخدمين بالتركيز على العمل الفعلي بدلاً من التفاصيل التقنية المعقدة.**

**4. Support for Non-linear Development (دعم للتطوير غير الخطي): Git يسمح بفتح الفروع (branches) والانتقال بينها ودمجها بسهولة، مما يسمح بالتطوير غير الخطي، حيث يمكن لفريق العمل العمل على ميزات مختلفة في وقت واحد دون تداخل.**

**5. Scalable (قابلية التوسع): Git قابل للتوسع ليتناسب مع احتياجات المشاريع الصغيرة والكبيرة على حد سواء، مما يجعله مناسبًا لمختلف أحجام الفرق والمشاريع.**

**باختصار، Git يوفر بيئة تطوير فعالة ومرونة للمطورين من خلال دمج ميزات مثل التوزيع والسرعة والبساطة ودعم التطوير غير الخطي والقابلية للتوسع.**

**Architecture in Git:**

الهندسة المعمارية في Git تشير إلى التصميم والبنية الداخلية لنظام إدارة النسخ الموزعة هذا.

هذا يشمل كيفية تنظيم البيانات والعمليات والتفاعلات داخل Git.

هذا التصميم يتيح لها توفير العديد من المزايا التي نعرفها، مثل التوزيع والسرعة والمرونة.

**هنا بعض الجوانب المهمة لهندسة Git:**

**1. نموذج البيانات في Git:**

- Git يستخدم نموذج بيانات مُسلسلة (Serialization Model) لتخزين المعلومات. هذا يعني أنه يتم تخزين كل شيء في قاعدة بيانات مركزية مع تسلسل (تسمى ذاكرة التخزين المؤقتة) لكل حالة في مراحل تطورها.

- تتكون العناصر الأساسية في نموذج البيانات من الـ "Blob" (تمثل الملف) والـ "Tree" (تمثل مجلد) والـ "Commit" (تمثل نقطة في التاريخ) والـ "Branch" (تمثل فرع من التطوير).

**2. نموذج التخزين (Storage Model):**

- Git يستخدم نموذج تخزين مضغوط مضغوط ذي كفاءة عالية. يتم تخزين الملفات والمعلومات في مجلد .git الخاص بكل مستودع.

- هذا النموذج يسمح بسرعة الوصول وفعالية الاستهلاك من حيث المساحة.

**3. التفاعل مع الـ "Remote Repositories":**

- Git يتيح التفاعل مع مستودعات بعيدة الموقع (remote repositories) بسهولة. يمكن للمستخدمين سحب (pull) التحديثات منها ودفع (push) التغييرات إليها.

- هذه العمليات تسهل التعاون بين المطورين والعمل على المشاريع المشتركة بشكل فعال.

**4. نظام الفروع (Branching System):**

- يُعتبر نظام الفروع في Git أحد أهم ميزاته. يتيح هذا النظام للمطورين العمل في بيئة منعزلة (فرع) واختبار التغييرات قبل دمجها في الفرع الرئيسي (master).

- هذا يسمح بتطوير غير خطي وتجربة الميزات دون التأثير على الإصدارات الأساسية.

**5. نظام العلامات (Tagging System):**

- يسمح Git بوضع علامات (tags) على نقاط محددة في التاريخ لتمييزها بشكل دائم.

- يستخدم ذلك لتحديد الإصدارات (releases) أو النقاط المهمة في التطوير.

Sha in git :

هحكي قصة لذيذه فشخ عن انواع الملفات والكلام الجميل دا :

دلوقتي احنا هنتكلم عن ال files علي انها blob طب اي السبب ولنفترض يسيدي انك الوقتي عندك ملف اسمه a.text انت مسحت الملف دا اصلا وضفت ملف تاني بنفس الاسم ازاي git يقدر يعرف الفرق ؟ فهنا احنا غيرنا اسمه من file ل blob والسبب بكل بساطة انه بقا هنا بيتعامل مع الملف علي اساس meta data بتاعت الفايل بقا بيتعامل معاه حرفيا علي اساس انه object اليل هو بص يمعلم انا عاوز اعرف الفايل دا مكوناته اي مش بس اسمه ايه الي هو استخدم ال sha يعني هاتلي الاتي الي هو عبارة عن الميتا داتا زي اي ؟ size , type وغيره فبقا هنا يقدر ياخد sha1 ويخزنه عنده بعدين يجي يبص علي التغيير الي انت عملته الي هو ال sha 1 بردو ويشوف هل في مسطره او سيمي كولون او سطر زيادة اولا ؟ بنائا علي هل ال sha1 === sha2 وهكذا بقا طيب عمالين نقول sha sha اي الشا دا يعم انت الشا دا بكل بساطة عامل زي دالة كدا بتاخد منك محتوي معين وليكن كلمة hello world يدخلها جوا دالة الدالة دي كل وظيتفها انها تطلعلك حاجة كدا زي باسورد او كلام متخلبط abdjekj233 وعدد الحروف دي بيكون حوالي اربعين حرف بالظبط ونفس الكلمة دي لو جيت كتبتها في اي مكان او كررتها هتطلعلك نفس الشا

## \_\_ شرح أنواع الملفات في Git:

**مقدمة:**

في Git، يتم التعامل مع جميع أنواع الملفات بشكل مختلف.

**الملفات كـ Blob:**

يتم التعامل مع الملفات كـ Blob، بمعنى أنها تُعامل كأشياء ثنائية (Binary Objects) وليست مجرد أسماء.

**لماذا؟**

* **التعرف على التغييرات:**
  + عند إجراء تغيير على ملف، مثل إضافة أو حذف سطر، يتم حساب قيمة SHA-1 جديدة للملف.
  + تقارن Git قيمة SHA-1 الجديدة بالقيمة القديمة لتحديد ما إذا كان الملف قد تم تغييره.
  + هذا يضمن تتبع Git للتغييرات بدقة، حتى لو تم تغيير اسم الملف.
* **المحتوى وليس الاسم:**
  + تركز Git على محتوى الملف، وليس فقط اسمه.
  + يسمح ذلك لـ Git بالتعرف على الملفات المتطابقة حتى لو كانت تحمل أسماء مختلفة.
* **الميتاداتا:**
  + بالإضافة إلى المحتوى، تخزن Git بعض المعلومات الإضافية عن الملف، مثل الحجم ونوع الملف.
  + تُستخدم هذه المعلومات لتحسين أداء Git ومساعدتها على تتبع الملفات بشكل أفضل.

**مثال:**

* لنفترض أن لديك ملفًا اسمه "a.txt".
* تقوم بحذف هذا الملف وإنشاء ملف جديد بنفس الاسم.
* من وجهة نظر Git، هذان ملفان مختلفان.
* لماذا؟
  + لأن محتوى الملفين مختلف.
  + ستقوم Git بحساب قيمة SHA-1 جديدة للملف الجديد، والتي ستختلف عن قيمة SHA-1 للملف القديم.

**SHA-1:**

* SHA-1 هي دالة تجزئة (Hash Function) تأخذ نصًا كمدخل وتنتج قيمة فريدة مكونة من 40 حرفًا.
* تُستخدم قيمة SHA-1 لتحديد هوية الملف بشكل فريد.
* حتى لو كان محتوى الملفين متشابهًا، فمن غير المحتمل أن يكون لديهما نفس قيمة SHA-1.

**مثال:**

* لنفترض أن لديك ملفًا اسمه "b.txt" يحتوي على النص "Hello, World!".
* تقوم بإنشاء ملف جديد اسمه "c.txt" بنفس المحتوى.
* ستقوم Git بحساب قيمة SHA-1 لكلا الملفين، وستكون نفس القيمة.
* هذا لأن محتوى الملفين متطابق.

**ملخص:**

* تُعامل Git الملفات كـ Blob، مما يعني أنها تُعامل كأشياء ثنائية.
* يتم استخدام قيمة SHA-1 لتحديد هوية الملف بشكل فريد.
* تتيح هذه الطريقة لـ Git تتبع التغييرات بدقة والتعرف على الملفات المتطابقة حتى لو كانت تحمل أسماء مختلفة.

**ملاحظة:**

* تم استبدال SHA-1 بـ SHA-256 في Git 2.25.
* ومع ذلك، لا يزال من الممكن استخدام SHA-1 مع Git.

**أمثلة إضافية:**

* يمكنك استخدام الأمر git hash-object لحساب قيمة SHA-1 للملف.
* يمكنك استخدام الأمر git log لعرض تاريخ التغييرات في الملف، بما في ذلك قيم SHA-1 لكل تغيير.
* يمكنك استخدام الأمر git diff لمقارنة محتوى الملفين.

مثلا :



بالشكل دا كدا تقدر تفهم وتلاحظ   
ولاحظ هنا انه عشان يطلعه زيه بالظبط اضاف تلت حاجات   
1 – ال type

2- ال size وبيضيف رقم زيادة علي الحجم عشان دا بيعمل بريك لاين لوحده في الاخر بس مش بيكون باين يعني

3- null character بجانب كل دا عشان كلمة hello world === hello world ودي فكرة ال sha ببساطة عشان كدا بيعرف بقا انت غيرت حاجة او لا طبعا كل دا بيتم في الخلفية

شوية اوامر عشان لو عاوز تعرف الحاجات في المشروع بتاعك عاملة ازاي

1- Work Tree : ls

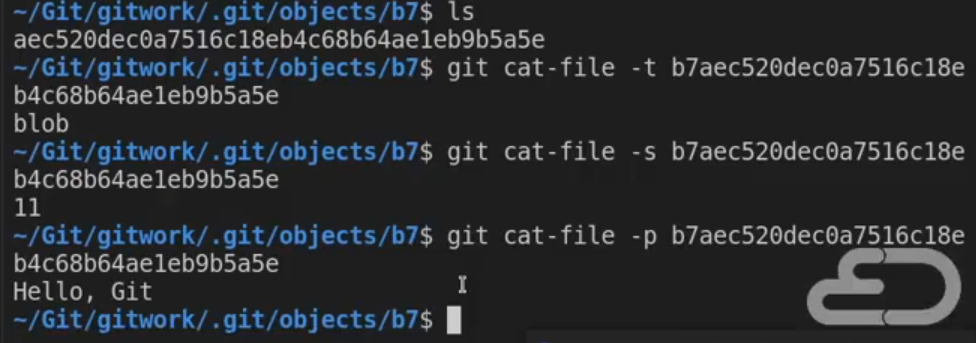
2-

- stage Area : git ls-files

- stage Area : git ls-files -s دا بيعرض الشا والبريميشن واسم الفايلات

3- git r epo : find .git/objects/ -type f

\_\_\_

اوامر لمعرفة نوع الملف حجم الملف محتويات الملف عن طريق ال Sha الخاص بالفايل لانه ممكن يكون tree , blob وهكذا انت مش هتكون عارف هو فولدر او فايل عن طريق ال sha   


في هنا معلومة عميقة شوية :

الوقتي لما انت بتعمل commit الي بيحصل دلوقتي انه بيروح يعمل تلاته object

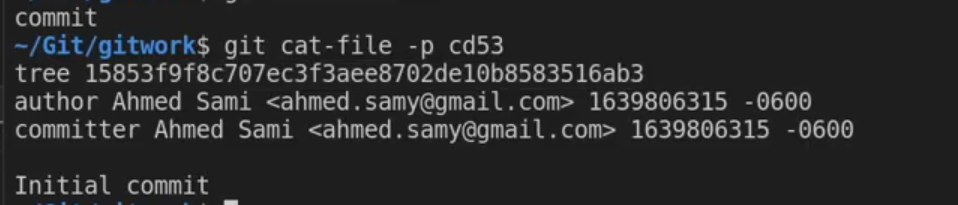
1- blob:

الملف الي اتعدل اصلا sha because it’s a object

2- tree:

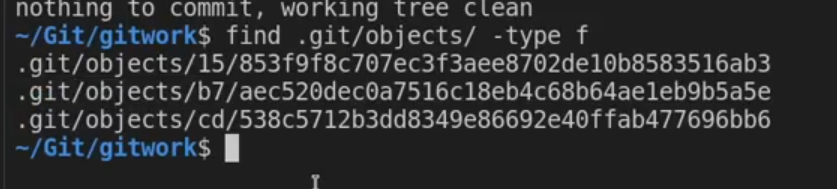
المكان او المسار الخاص بالفايل المتعدل sha because it’s a object

3- commit:

بيكون في داخله شا لتري لو هو اول كوميت فهوا ليه او بداخله التري او الفولدر دا وبيكون حاططلك الشا جمبه تقدر تكتب الامر الخاص بعرض المحتوي وتشوف المحتويات  


الرابر او الشئ الي بيقولك فلان الفلاني الي اسمه كذا وايميله كذا والكونفيج بتاعه كذا عمل كذا كذا تحت مسمي انه اصدرا واحد يعني ببساطة التعديل الي تم في كل الاوبجيكتس دي الشخص دا الي عملها في هيئة التعديل او الكوميت دا sha because it’s a object

لاحظ لما نقول ابوجيكت تفهم ايه ؟؟؟؟؟؟؟؟؟ **انه ليه حاجة اسمها Sha**



فهتلاقي المنظر دا كدا برغم اني عامل commit واحد وعندي فايل واحد اصلا

**عشان تقدر تحدد نوع ال Sha**   
git cat-file -t sha

**عشان تقدر تحدد حجم ال Sha**git cat-file -s sha

**عشان تقدر تجيب محتوي ال Sha**

Git cat-file –p sha

**ركز هنا :**

1- محتويات commit بتكون عبارة عن المالك والتاريخ وال sha وال tree

2- محتويات ال tree بتكون الفايل اسم الفايل بال sha

3- محتويات ال blob بتكون زي مثلا “hello git"

**طبعا دي كانت الطريقة البلدي لعرض محتويات وتفاصيل الكومميت**

**تقدر تستخدم حاجة اشيك زي**   
git show “sha of commit”

**يبقا نستنتج ان اي commit هتعلمه هيحفظ تلاته object اقل حاجة حتي لو غيرت حرف واحد جوا فايل واحد .**

ملاحظات للامر Git diff

1- لو كتبت git diff لوحدها كدا فانا كدا بطالب بالفرق بين working tree, index الي هو ال stage

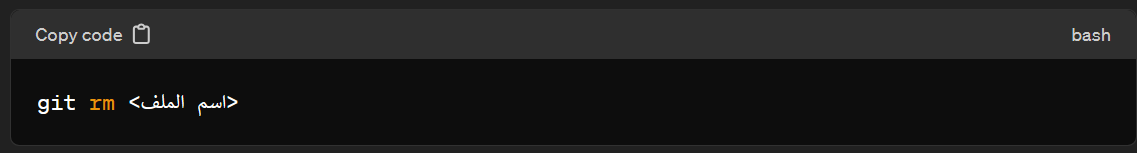
2- git diff –staged دا بيجبلي الفرق بين الريبو وال stage area

لمعرفة الفرق بين اتنين كومميت نعمل اي ؟

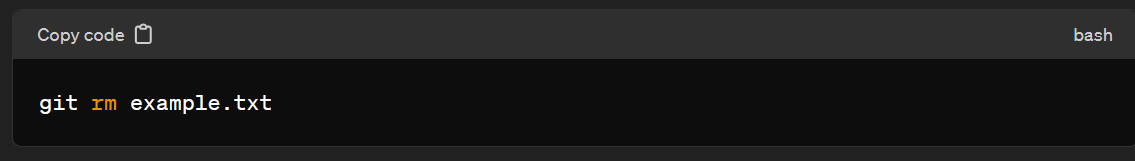
Git diff shacommit1..shacommit2

اوامر الحذف في git :

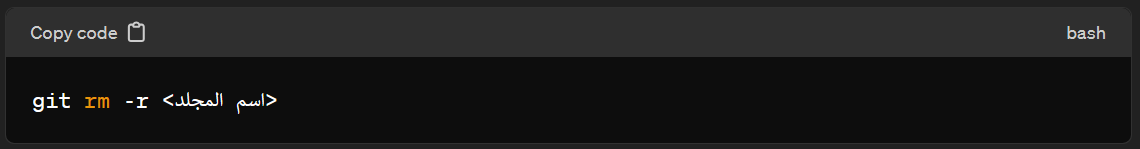
### حذف ملف معين:

 يستخدم هذا الأمر لحذف ملف محدد من مشروع Git.

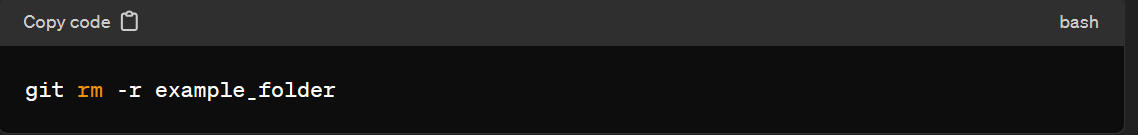
مثال:



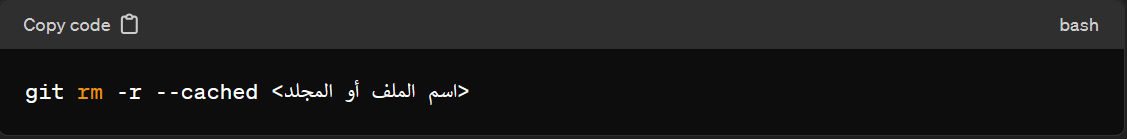
### حذف مجلد (فولدر):

 يستخدم هذا الأمر لحذف مجلد كامل مع كل محتوياته.

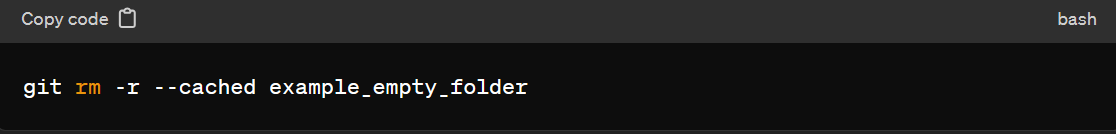
مثال:



### حذف ملفات ومجلدات فارغة (Empty):

 يستخدم هذا الأمر لحذف ملفات أو مجلدات فارغة من النسخة المحلية للمشروع دون أن يؤثر على الملفات في المستودع البعيد.

مثال:



الفرق بين git reset، git revert، و git reset --hard يتعلق بكيفية التعامل مع تاريخ المشروع وإلغاء التغييرات.

1. git reset:

- يُستخدم لتغيير موضع HEAD (وبالتالي فرع العمل الحالي) إلى commit محدد.

- يُمكن استخدامه لإلغاء التغييرات المحلية المؤقتة بعد القيام بعمل commit.

- يمكن استخدامه لإزالة الملفات من مرحلة الاستعداد (staging area) دون حذفها من النسخة المحلية.

- يُستخدم بشكل رئيسي لتغيير تاريخ المشروع المحلي.

2. git revert:

- يُستخدم لإنشاء commit جديد يُلغي التغييرات التي تمت في commit محددة مسبقًا.

- لا يُحذف أي commit سابقة، بل يُنشئ commit جديد يُقوم بإلغاء تأثير التغييرات التي تمت في commit سابقة.

- يُستخدم بشكل آمن لإلغاء التغييرات المرتبطة بـ commit سابقة دون تأثير على تاريخ المشروع.

3. git reset --hard:

- يُستخدم لإعادة HEAD وفهرس العمل (working directory) ومرحلة الاستعداد (staging area) إلى حالة معينة محددة بواسطة commit.

- يُمكن استخدامه للرجوع إلى commit سابق بشكل قاطع، وبالتالي فإنه يحذف التغييرات التي تمت بعد ذلك ال commit.

- يجب استخدامه بحذر لأنه يؤدي إلى فقدان التغييرات غير المحفوظة في مرحلة الاستعداد وفهرس العمل.

باختصار، git reset تغيير موضع HEAD والتعامل مع تاريخ المشروع، git revert ينشئ commit جديد لإلغاء تأثير commit سابقة، أما git reset --hard فيحذف التغييرات المحلية ويعيد HEAD وفهرس العمل إلى commit محدد.

أوامر ال Git Log

طبعًا! إليك قائمة بأهم أوامر git log وتفاصيلها وأهميتها:

1. git log: تعرض سجل التاريخ الكامل للتعديلات على المشروع. تظهر كل تعديلات الشجرة مع تفاصيلها مثل مؤلف التعديل، تاريخه، ورسالة الالتزام(commit message).

2. git log --oneline: يعرض قائمة مختصرة لكل التعديلات في سجل التاريخ، كل تعديل في سطر واحد فقط مع الرقم التسلسلي للتعديل ورسالة الالتزام.

3. git log --graph: يعرض سجل التاريخ بتخطيط الشجرة، مما يسهل رؤية التفرعات والاندماجات.

4. git log --author=author: يقوم بتصفية سجل التاريخ لعرض التعديلات التي تم إجراؤها بواسطة المؤلف المحدد.

5. git log --grep=pattern: يبحث عن النص المحدد في رسائل الالتزام(commit messages) ويعرض التعديلات المطابقة.

6. git log --since=date و git log --until=date: يعرضان التعديلات التي تم إجراؤها بين تواريخ محددة.

7. git log file: يعرض التاريخ الخاص بملف محدد فقط.

8. git log --stat: يعرض إحصائيات مختصرة توضح التغييرات في كل تعديل، مثل عدد الإدراجات والحذفيات.

9. git log -p أو git log --patch: يعرض التغييرات الفردية (diffs) لكل تعديل، مما يوفر نظرة مفصلة على ما تم تغييره.

10. git log --graph --oneline --decorate --all: يعرض سجل التاريخ مع تخطيط الشجرة والرموز المميزة لكل فرع.

هذه الأوامر تساعدك على استعراض تاريخ المشروع بطرق مختلفة وفقًا للاحتياجات الخاصة بك، سواء كنت ترغب في معرفة من قام بتغيير معين أو مراجعة التغييرات في فترة زمنية معينة أو فهم تفاصيل التعديلات في ملف محدد.

أمر ال Git reflog

أمر git reflog يستخدم لعرض سجل الإشارات (Reference Log) في مشروع Git. هذا الأمر يعرض سجلًا لجميع العمليات التي تمت على الإشارات في مشروعك، بما في ذلك التفاعلات مع الفروع (branches) والعلامات (tags) وغيرها من الإشارات المحلية. يُعتبر git reflog أداة مفيدة لتحديد وتصحيح الأخطاء، مثل فقدان التاريخ أو التحرك بطريقة غير متوقعة بين الفروع.

هنا بعض التفاصيل حول أهم الاستخدامات والخيارات الشائعة لأمر git reflog:

git reflog: يعرض قائمة بجميع العمليات التي تمت على الإشارات مع تواريخها والهاش الخاص بها.

git reflog show reference: يعرض سجل الإشارات الخاص بالإشارة المحددة (مثل اسم الفرع أو العلامة).

git reflog expire: يقوم بحذف الإدخالات القديمة في سجل الإشارات، مما يساعد في تنظيفه وتقليل حجمه.

git reflog delete reference: يقوم بحذف سجل الإشارات الخاص بالإشارة المحددة.

سجل الإشارات (Reference Log) يتيح لك فهم تاريخ المشروع بشكل أفضل وتتبع العمليات التي قمت بها على الإشارات، مما يساعد في تحديد الأخطاء وإصلاحها واستعادة التغييرات المفقودة.

Git diff

1. git diff: يعرض التغييرات التي لم تُنفذ بعد (غير المنطبقة) بين مرحلة العمل (ملفات العمل الحالية) ومرحلة الانتظار (التعديلات التي تمت إضافتها إلى مرحلة الانتظار).

2. git diff commit: يعرض التغييرات بين مجلد العمل الحالي والتعديلات الموجودة في commit المحدد.

3. git diff commit1 commit2: يعرض التغييرات بين اثنين من commits.

4. git diff --cached: يعرض التغييرات التي تمت إضافتها إلى مرحلة الانتظار مقارنة بالمرحلة الأخيرة في مستودع Git.

5. git diff --staged: يعرض التغييرات الموجودة في مرحلة الانتظار

(مشابهة لـ git diff --cached).

6. git diff --name-only: يعرض قائمة بأسماء الملفات التي تم تغييرها دون عرض التفاصيل.

7. git diff --stat: يعرض إحصائيات مختصرة للتغييرات، مثل عدد الملفات المعدلة وعدد الإدراجات والحذفيات.

8. git diff branch1..branch2: يعرض التغييرات بين اثنين من الفروع.

9. git diff --color-words: يظهر التغييرات بين الكلمات بلون مختلف لتسهيل القراءة.

10. git diff HEAD: يعرض التغييرات بين مجلد العمل الحالي وآخر commit.

تُستخدم هذه الأوامر لمراجعة التغييرات في مشروع Git، مما يمكنك من فهم التعديلات التي تم إجراؤها والتحكم في عملية التطوير وإدارة التغييرات.

اوامر ال Git show

1. git show: هذا الأمر يُظهر تفاصيل التعديل الأخير في الفرع الحالي. يتم استخدامه بدون أي معاملات لعرض آخر commit الذي تم إجراؤه.

2. git show commit: يعرض تفاصيل commit معينة بواسطة تحديد هويتها (hash). يمكن استخدام القيمة التي تمثلها SHA-1 hash أو الرقم التسلسلي لل commit.

3. git show commit file: يعرض التغييرات التي تم إجراؤها على ملف معين في commit محدد.

4. git show HEAD: يعرض تفاصيل التعديل الأخير في الفرع الحالي، مماثل ل git show بدون أي معاملات.

5. git show --name-only commit: يعرض قائمة بالملفات التي تم تغييرها في commit محدد دون عرض التفاصيل.

6. git show --stat commit: يعرض إحصائيات مختصرة للتعديلات التي تم إجراؤها في commit محدد، مثل عدد الملفات المعدلة وعدد الإدراجات والحذفيات.

7. git show --patch commit: يعرض التغييرات بشكل مفصل لكل commit، مع توضيح الإضافات والحذفيات في كل ملف.

8. git show --graph commit: يعرض تاريخ commit محدد بتخطيط الشجرة لتوضيح العلاقات بين الفروع والاندماجات.

9. git show --pretty=format commit: يسمح بتخصيص تنسيق عرض التفاصيل لكل commit وفقًا لقالب محدد، مثل --pretty=oneline لعرض التفاصيل في سطر واحد لكل commit.

هذه الأوامر تُستخدم لفهم التغييرات في مشروع Git، سواء كنت تحتاج لمراجعة التعديلات في commit محدد، أو لتحليل تفاصيل معينة في ملفات، أو لفهم السجل الزمني للمشروع بشكل عام.

**Branches**

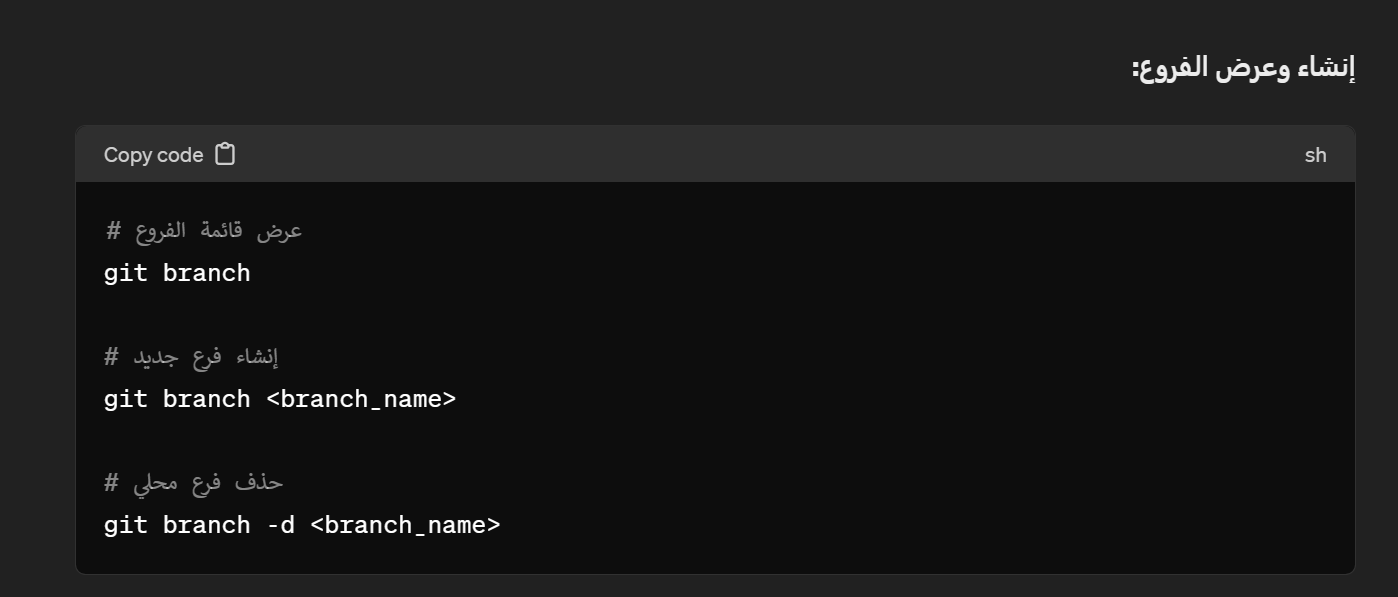
1- مكونات الBranches البرانش عبارة عن مجموعة من ال Commits

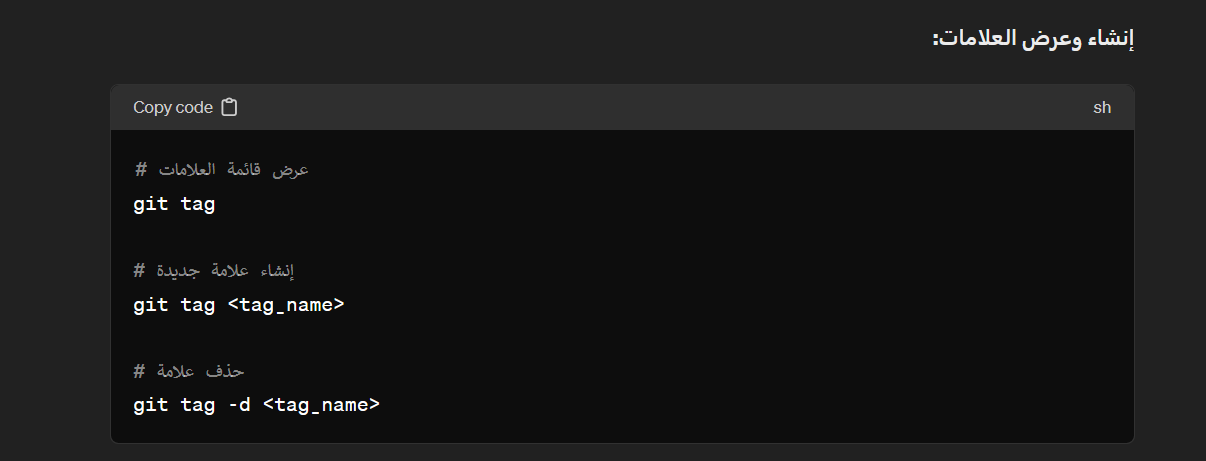
2- عندك حاجة اسمها تاج الي بتخليك تقدر تقسم المشروع بتاعك لمجموعة من ال versions ببساطة يعني بعد مثلا عشرين كوميت انت قررت تطلع اول اصدار عندك من المشروع في الوقت دا بقا انت كدا يمعلم بتقرر انك تعمل حاجة اسمها Tag .

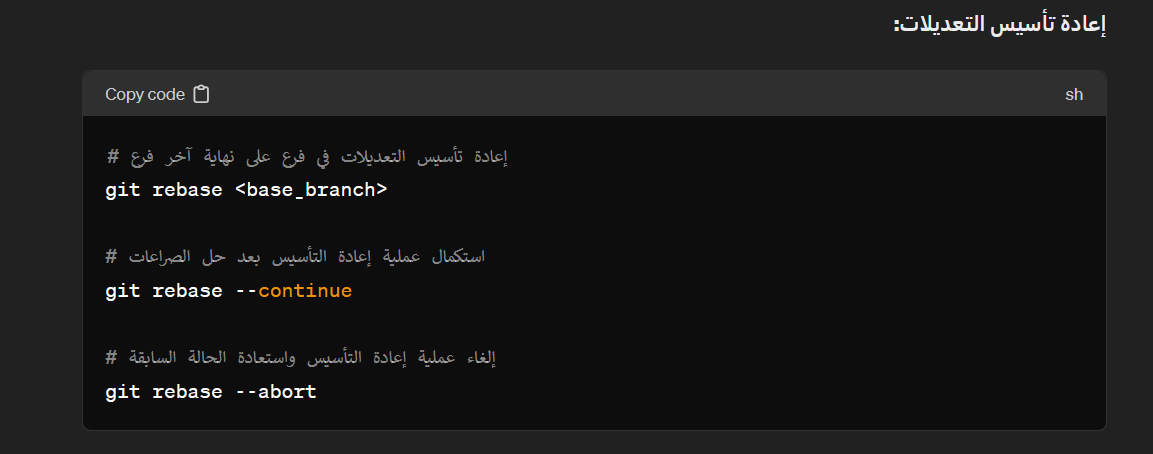
3- عندك حاجة اسمها git rebease وظيفتها انك بكل بساطة يبرنس لما بتاخد برانش فرعي من الماستر ومثلا بتعمل عليها تعديلات معينة وليكن في الماستر هتروح تضيف ملف جديد , وفي البرانش الفرعي الي انت واخده من الماستر وليكن اسمها testing انت جيت عدلت فيه بردو فانت كدا عندك الموضوع تشعب فانت محتاج الوقتي لما تيجي تعمل merge انك هتضيف كوميت جديد خالص بحيث بعد التشعب دا يروحوا يلتقوا في نقطة جديدة تماما لكن شكل الكوميتس هيكون متشعب زي ما هو طالما هتعمل ال merge دا بأمر ال merge العادي ولكن لو عملته بال rebaise هيكون الشكل الخاص بال commits شكل خطي مش متشعب .

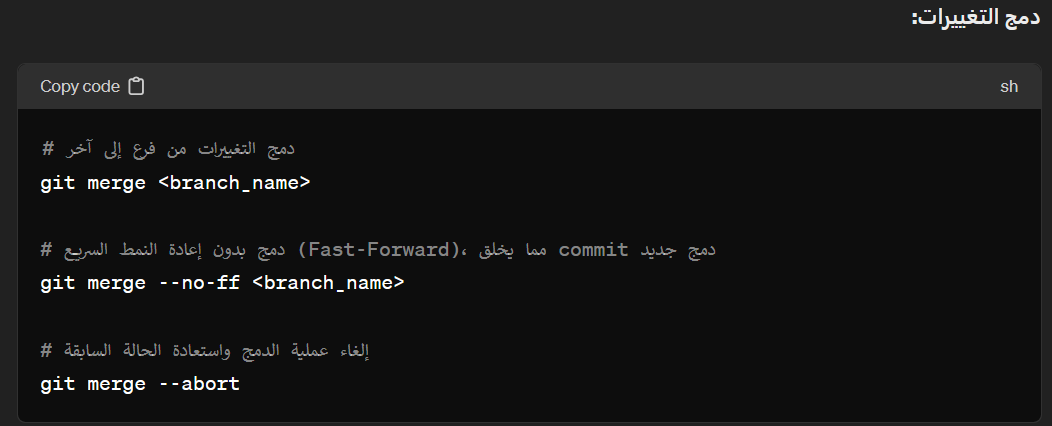
4- لاحظ ان لو انت مثلا واقف علي الماستر وشغال في فايل اسمه index.html مثلا وقمت واخد برانش جديد وروحت عدلت في ال index.html دا في الملف الجديد جميل جميل وقبل ما تعمل merge مع الماستر قررت انك تعدل في ال index.html ولكن في الماستر فلما تيجي تعمل merge هيحصل conflict بحيث هيكون في تضارب بين ال master , testing brach فانت بتفتح الملف وتقرر اي تعديلات انت حابب تسيبها واي تعديلات انت حابب تشيلها انت وحسب رؤيتك بقا .

اهم الاوامر بناءا علي هذه الفقرة :









**الاوامر الخاصة بال Remote repo**

1. git remote: يعرض قائمة بجميع الـ Remote الموجودة في المشروع.

2. git remote -v: يعرض قائمة بجميع الـ Remote الموجودة في المشروع مع عناوين URL الخاصة بها.

3. git fetch remote: يستخدم لجلب التحديثات الجديدة من الـ Remote المحدد.

4. git pull remote branch: يستخدم لجلب التحديثات الجديدة من الـ Remote المحدد ودمجها مع الفرع المحلي.

5. git push remote branch: يستخدم لدفع التحديثات المحلية إلى الـ Remote المحدد.

6. git branch -r: يعرض قائمة بجميع الفروع البعيدة (Remote Branches).

7. git checkout -b local\_branch remote/remote\_branch: ينشئ فرعًا محليًا جديدًا ويتبع الفرع البعيد.

8. git remote add name url: يضيف Remote جديد بالاسم المحدد والعنوان URL.

9. git remote remove name: يقوم بإزالة Remote المحدد من المشروع.

