Missing Semester Lectures

MIT - Jan 2020

Lecture 6: Version Control (git)

Waseem Kntar Notes 12/04/2020

مقدمة:

تتكلم هذه المحاضرة عن نظام إدارة نسخ الكود (VCS) وذلك بالاعتماد على النظام المفتوح المصدر git، حيث يعد العمل على هذا النظام مفيد جداً في العمل التعاوني على مشروع (Collaboration)، حفظ النسخ القديمة من الكود وحتى حفظ الكود نفسه لذلك هي مفيدة حتى مع العمل الفردي أيضاً، وتساعد كذلك في الإجابة على عدّة أسئلة مهمّة مثل:

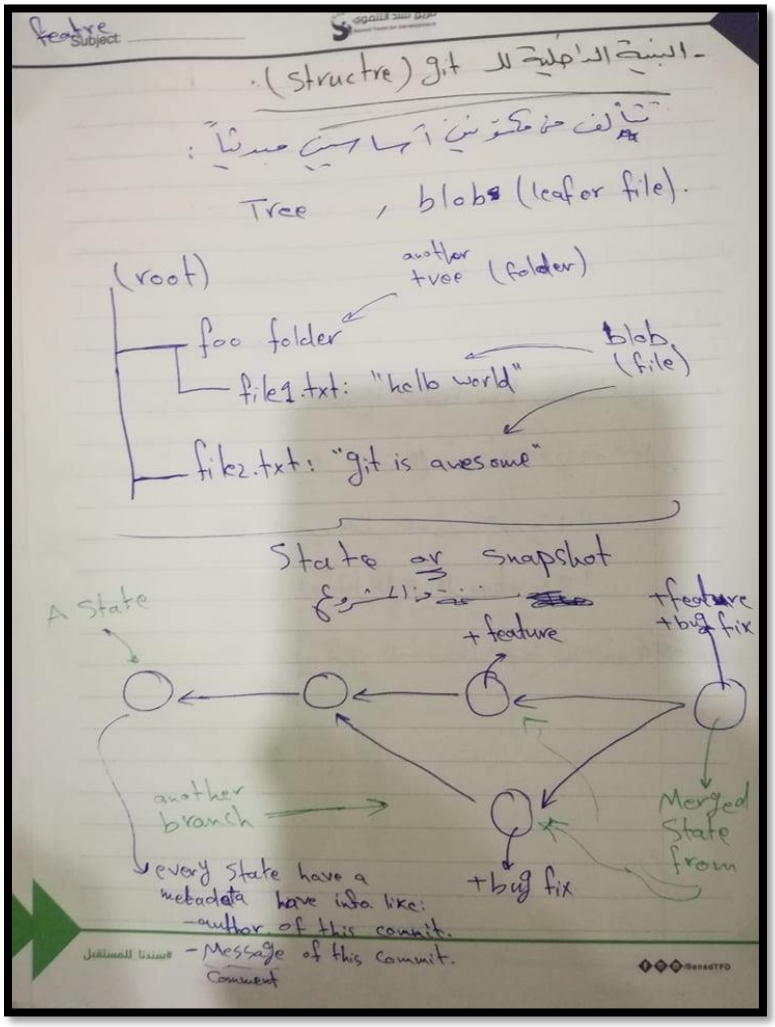
- "مين عدّل هون؟"
- "مين اشتغل على هي الـ feature بدنا نسألوا عن شغلة؟"
- "بدي أرجع للنسخة الماضية من كودي"
- ...

> Because Git's interface is a leaky abstraction, learning Git top-down (starting with its interface / command-line interface) can lead to a lot of confusion. It's possible to memorize a handful of commands and think of them as magic incantations, and follow the approach in the comic above whenever anything goes wrong.
>
> While Git admittedly has an ugly interface, its underlying design and ideas are beautiful. While an ugly interface has to be *memorized*, a beautiful design can be *understood*. For this reason, we give a bottom-up explanation of Git, starting with its data model and later covering the command-line interface. Once the data model is understood, the commands can be better understood, in terms of how they manipulate the underlying data model.

اقتباس 1- اقتباس يوضح هدف المحاضرة **[1]**

المحاضرة مميزة عن غيرها بأنها تبدأ بشرح نموذج المعطيات المستخدم في الـ git وكيف يتم التخزين والبحث ومن ثم وبعد ذلك تنتقل إلى التعليمات الخاصة بالـ git، أي اهتموا بشرح الـ core الذي بُني عليه الـ git على عكس الكثير من محتويات أخرى التي تدخل فوراً إلى التعليمات وحتى في كثير من الأحيان دون شرح عنها أو كيف تعمل من الداخل.

نموذج المعطيات في الـ git (git Data Model):



- Git Data structure:

```
1. // a file is a bunch of bytes
2. type blob = array<byte>
3.
4. // a directory contains named files and directories
5. type tree = map<string, tree | file>
6.
7. // a commit has parents, metadata, and the top-level tree
8. type commit = struct {
```

```
9.      parent: array<commit> //array of parent because a commit can have
   more than one parent in case of merge commit
10.    author: string
11.    message: string
12.    snapshot: tree
13.  }
```

- But how previous data are stored on disk?

الفكرة أنّه من أجل تأمين عملية البحث السريع (تعقيد زمني أقل) وتخزين أقل (ذاكرة أقل)، فإنّه تم الاعتماد على مايلي:

```
1. type object = blob | tree | commit
2. objects = map<string, object> //string is the hash function of object
3.
4. def store(object):
5.     id = sha1(object)
6.     objects[id] = object
7.
8. def load(id):
9.     return objects[id]
```

إذاً، البحث يتمّ عن طريق فقط الهاش الخاص بالأوبجكت وبالتالي سرعة أكبر وهذا مايُسمى بالـ Hash Content address (SHA-1)، وكما نعلم أن أن الـ Hash function هو تابع يوّلد قيمة unique لدخله وبالتالي يمكن استخدامه كـ id.

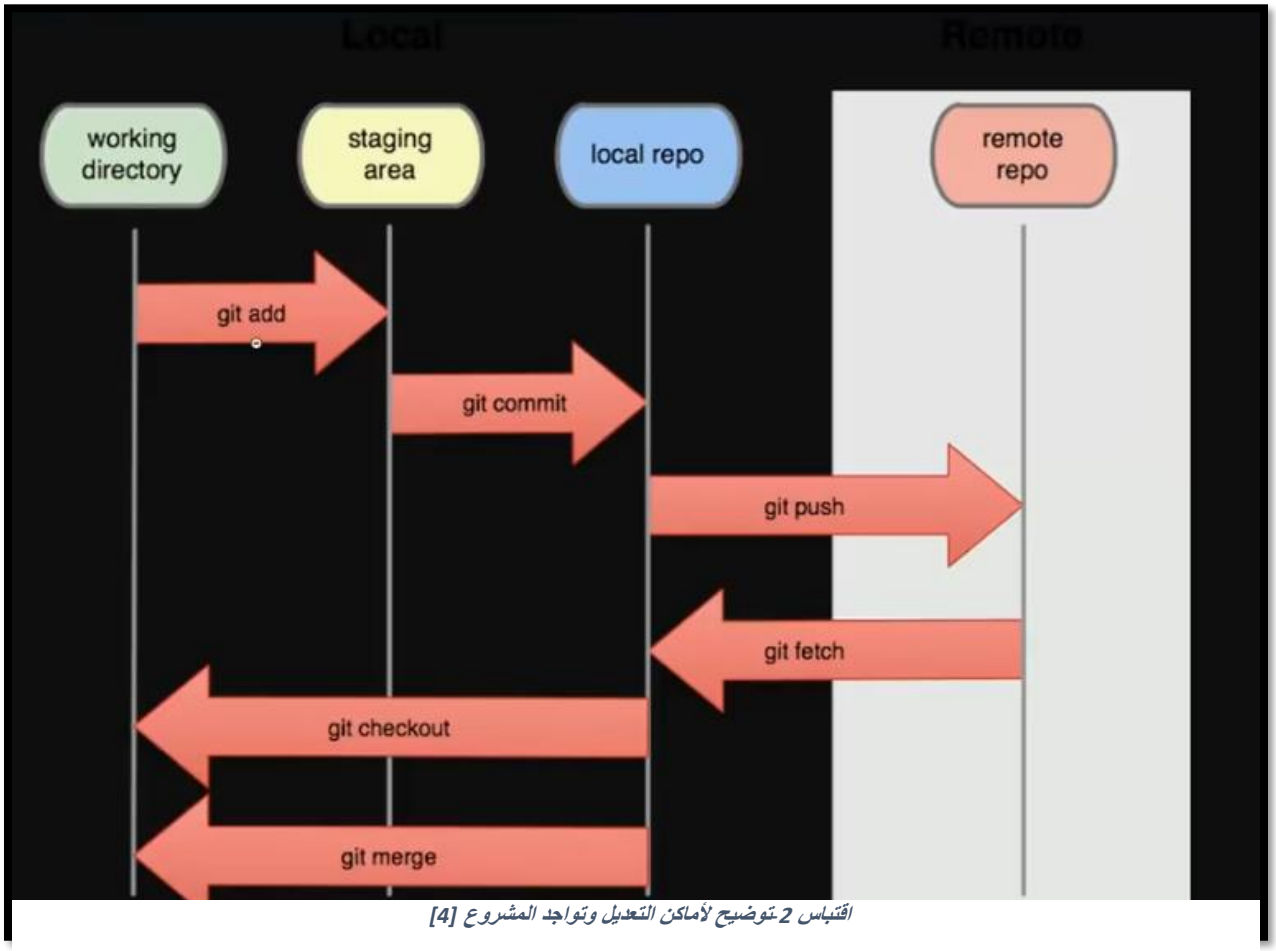
ولكن.. السؤال الآن أنّه إذا أردنا الوصول لـ Commit معيّن من خلال اسمه (الذي كان Waseem commit مثلاً)، كيف يكون الوصول والبنية التي لدينا هي Objects تحتوي الهاش كمفتاح والأوبجكت كvalue؟؟

لذلك كان مايُسمى بالـ reference وهي Map المفتاح فيها هو human readable string والـ value هي الهاش.

```
1. references = map<string, string> // key is a human readable string,
   value is the hash
2.
3. def update_reference(name, id):
4.     references[name] = id
5.
6. def read_reference(name):
7.     return references[name]
8.
9. def load_reference(name_or_id):
10.      if name_or_id in references:
11.          return load(references[name_or_id])
12.      else:
13.          return load(name_or_id)
```

وبالتالي أصبح بالإمكان تسمية snapshot معيّنة باسم نفهمه (مثل master) عوضاً عن الهاش غير المفهوم.

- شكل معبّر عن أماكن تواجد المشروع والحالات التي يمر بها:



اقتباس 2 توضيح لأماكن التعديل وتواجد المشروع [4]

- Let's play with the practical part:

1- تعليمات عامّة:

Command التعليمة	عملها
`git init`	تعليمة ننفذها لمرة واحدة في البداية في مجلد المشروع وهي لتهيئة ملفات الـ git.
`echo "hello world git" > hello.txt`	التهيئة بالجملة "hello world git" للملف النصي وإنشاؤه إذا لم يكن منشأ.
`git status`	تُعطي معلومات عن حالة المشروع والتغييرات الحاصلة إن وجدت.
`git add <filename>`	إضافة <filename> إلى الـ staging area، وفي حال نفذنا git status الآن سيطبع بأن هناك ملف ينتظر عمل commit له.
`git add :/`	إضافة جميع التغيرات الحاصلة إلى الـ staging area.
`git commit`	رفع التغيرات في الـ state الحالية إلى الـ local repo.
`git commit -m "message"`	نفس الماضية ولكن مع رسالة توضيحية آنية بدون الحاجة إلى فتح المحرر كما في الماضية.
`git commit -a`	رفع جميع التغيرات.

`git log`	Visualize commits history.
`git log --all --graph --decorate`	إظهار الـ history ولكن بشكل غراف.
`git checkout <revision>`	ينقلني إلى الـ commit المراد (revision == commit hash) أي ينقل المؤشر HEAD إلى هذا الـ commit وهذا المؤشر يؤشر دائماً على الحالة الحالية التي نحن فيها.
`git checkout <filename>`	تجاهل التغييرات الحاصلة على الملف في الـ working dir.
`git diff <filename>`	إظهار التغييرات على الملف التي حدثت في الـ commit الحالي.
`git diff <revision> <filename>`	إظهار التغييرات التي حصلت منذ commit معيّن.
`git diff <revision1> <revision2> <filename>`	إظهار التغييرات التي حصلت منذ الـ revision1 حتى الـ revision2.

2- تعليمات Branching & Merging:

التعليمة	عملها
`git branch`	إظهار الـ branches.
`git branch <branch-name>`	إنشاء فرع جديد ويكون بداية في الـ commit الحالي.
`git checkout <branch-name>`	جعل الـ HEAD يؤشر على هذا الـ branch، وبالتالي أي commit قادم سيكون من هذا الـ branch.
`git merge <branch-name>`	دمج الفرع المُمرر مع الحالي، ويمكن تنفيذ دمج أكثر من فرع مع الحالي عبر: git merge <branch1> <branch2> وسينتج تضاربات (في حال تواجد تعديلات على نفس الأجزاء من الكود) وحلها يكون بشكل يدوي.
`git log --all --graph --decorate --oneline`	طباعة الـ history ولكن بسطر واحد لكل commit.

3- Git Remote:

التعليمة	عملها
`git init --bare`	فقط في حالة كان الـ remote على الـ local machine.
`git remote add <name> <url>`	إعلام الـ repo الحالي بالـ remote (نعطي اسم له ونضع الرابط).
`git fetch <remote-name> <from-branch>`	جلب مافي الـ remote إلى الحالة الحالية .
`git rebase -i <remote-name/from-branch>`	تطبيق التغييرات على الـ remote.
`git merge`	عند تطبيقها بعد الـ git fetch فإنها تنقل المؤشر HEAD إلى الـ remote.
`git pull <remote-name> <from-branch>`	تكافئ git fetch + git merge
`git clone <remote> <directory>`	جلب نسخة من الـ remote repo وإنشاء مجلد لها.
`git push <remote> <branch>`	رفع التغييرات إلى الـ <remote> من الفرع <branch> (وإذا كان الفرع غير موجود في الـ remote repo سيُنشأ تلقائياً).[5]

4- Important Commands:

التعليمة	عملها
`vim ~/.gitconfig`	تعديل بعض البارمترات مثل ..... user name, email
`git clone …. --shallow`	جلب آخر تغيير على الـ repo فقط.
`git add -p <filename>`	إضافة التغييرات بشكل تفاعلي (مثلاً في حالة وجود تعليمة أو سطر لانريد إضافته للـ staging area الآن مثل تعليمة طباعة لغرض الـ debugging مثلاً).
`git bisect`	تستخدم الـ binary search في البحث عن الـ commit التي فشل عندها الكود في الـ unit testing.
`vim .gitignore`	نستدعي هذه التعليمة في dir الـ repo ونكتب في الملف لواحق أو أسماء الملفات التي لانريد إشراكها في الـ commits أو الـ adds (التي نريد تجاهلها من الـ git بأكمله).

ملاحظات:

1- لاستخدام الـ git على الـ Local machine يمكنك تحميله من هنا: https://git-scm.com/downloads

2- من أجل الـ remote repos يمكنك استخدام مواقع مثل Github, gitlab, bitbucket، حيث إن Github يوجد فيه free plan لكنه محظور في سوريا (من غير الممكن إنشاء مشاريع private)، gitlab مدفوع ويوجد free plan بالإضافة إلى أنّه يوفر خيار تنصيب بيئة الـ gitlab على Local host، bitbucket مجاني تماماً ولكنه يحتاج vpn (محظور في سوريا).

3- يمكنك استخدام الـ software التي تستخدم نظام الـgit ولها gui جاهزة وذلك لتتجنب الـ command-line، مثل برنامج source tree، يمكنك تحميله من هنا: https://www.sourcetreeapp.com/

مراجع:

[1] Lecture video: https://youtu.be/2sjqTHE0zok

[2] Lecture Notes: https://missing.csail.mit.edu/2020/version-control/

[3] Why we are teaching this class? https://missing.csail.mit.edu/about

[4] Elzero Web School YouTube channel: Learn Git and Github.

[5] https://git-scm.com/docs/git-push#Documentation/git-push.txt-codegitpushoriginmastercode

وسيم قنطار
دمشق 12/04/2020