digital whisper
Technologic Papers

Git Internals

מאת אור (OrKa) קמארה

הקדמה

מי מאיתנו לא שמע על Git? כנראה שרובנו שמענו...

לאלה שלא שמעו - בטוח שמעתם על Source Controls אחרים. אז Git הוא עוד Source Control, ולא סתם אחד, כנראה הטוב ביותר, לא רק כי אני חושב את זה, אלא בעיקר על סמך מגמת השימוש העולה בו מאז שיצא לאוויר העולם.
במאמר זה נסביר בקצרה על Git ולאחר מכן נצלול לעומק, נלמד איך Git ממומש ואיך פקודות בסיסיות שאנחנו מבצעים בעבודה השוטפת שלנו עובדות מאחורי הקלעים.

אתם בטח שואלים למה זה חשוב להיכנס לקרביים של **Git**?

1. חשוב לדעת איך דברים שאתם משתמשים בהם עובדים
2. תוכלו להשתמש בידע הזה כשאתם מפתחים מערכות משלכם
3. זה מגניב! ממש מגניב!

אז מה זה בעצם Git?

Git היא מערכת ניהול גרסאות מבוזרת (Distributed Version Control System) אשר נוצרה בשנת 2005 על ידי לינוס טורבאלדס (כן כן, האיש מאחורי לינוקס). לאחר מספר נסיונות לאמץ מערכות שונות לניהול קוד המקור של לינוקס, הוא לקח את המושכות לידיים והחליט לפתח מערכת משלו.

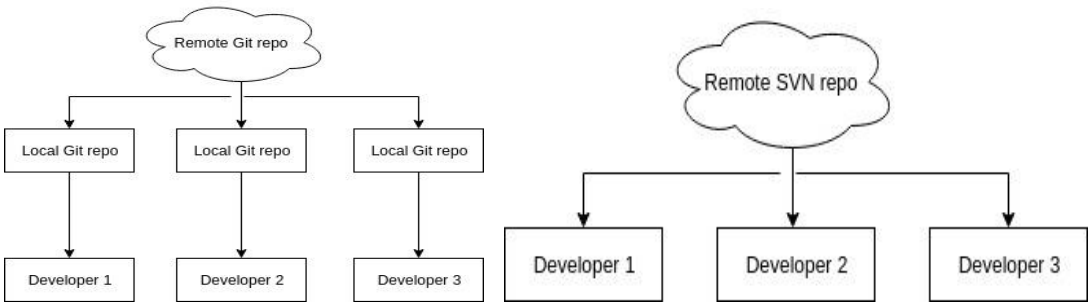
עובדה משעשעת: משמעות המילה git היא אדם לא נעים / ממזר. כאשר נשאל לינוס למה קרא למערכת בשם הזה אמר: "אני ממזר אנוכי, ואני קורא לכל הפרויקטים שלי על שמי. קודם Linux, ועכשיו Git."

מאז ועד היום Git הפכה למערכת ניהול הגרסאות הנפוצה ביותר בעולם. דוגמא לכך, ניתן לראות את כמות שירותי האחסון לניהול גרסאות אשר מבוססים על Git. החל מ-GitHub שנקנתה לאחרונה על ידי מיקרוסופט בסכום בדיוני של 7.5 מיליארד דולר (לא יכלו לעגל ל-8?!) BitBucket, GitLab, ואפילו TFS של מיקרוסופט (יכול לעבוד על בסיס Git או על בסיס מערכת שונה שפותחה על ידי מיקרוסופט).

הבדלים בין Git לבין Source Control אחרים

לפני שנתחיל לצלול לעומק ולדבר על מאחורי הקלעים של Git, נסביר בכמה נקודות את ההבדלים בינו לבין Source Controls אחרים (נבצע את ההשוואה ל-SVN):

1. כמו שנאמר מקודם, מדובר במערכת קבצים מבוזרת. כלומר, כאשר מריצים את הפקודה `git clone` מקבלים העתק מקומי ומלא של כל היסטוריית הפרויקט. לכל משתמש של של הפרויקט יש העתק מלא של ה-repository (מעכשיו נקרא לו repo) על המחשב שלו. זאת בניגוד ל-SVN שממומש על ידי repo אחד מרכזי, מה שמקל על פעולות כמו ניהול הרשאות, אבטחה או העתקה של כל ה-repo.



[שרטוט 1: ארכיטקטורת Git לעומת SVN]

2. בשל העובדה ש-Git דורש שכל ההיסטוריה של הפרויקט תהיה זמינה, גם אם המשתמש לא נגיש ל-repo המרכזי (כי אין לו חיבור לרשת למשל), הוא יכול לעבור בין שלבים שונים בהיסטוריית הפרויקט (לעבור בין commits). ב-SVN, התקשורת מתבצעת כאמור רק מול השרת הראשי, מה שמחייב את המשתמש לקבל רק גרסה אחת של הפרויקט בכל שלב נתון.
3. יתרון נוסף ל-Git הוא ה-staging area או קובץ ה-index כמו שנראה בהמשך. בכל ביצוע של שינוי בסביבת העבודה, עוברים בשלב נוסף המאפשר לפצל שינוי גדול שנעשה להרבה commits קטנים.

Flow בסיסי של Git

בחלק זה נסביר כמה פקודות בסיסיות של Git על מנת להבין את התהליך העבודה הכללי... בהמשך המאמר נסביר מה משמעות פקודות אלו.
אגב, חלק זה מיועד לאנשים ללא נסיון ב-Git - מי שכבר מנוסה בסביבה מוזמן לדלג עליו.

נחלק את העבודה ב-Git למספר שלבים:

1. **אתחול הסביבה:**

כאשר מתחילים לעבוד על פרוייקט, אנחנו צריכים להחזיק את ה-repo שלו בסביבת העבודה שלנו. לשם כך יש שתי אופציות:

a. `git init` - כאשר רוצים לייצר repo חדש מאפס
b. `git clone` - כאשר רוצים להעתיק repo מרוחק (קיים) לסביבת העבודה שלנו

2. **ביצוע שינויים בסביבה:**

לאחר כל שינוי של קובץ נרצה להוסיף אותו ל-staging area ולבצע commit לשינוי הרלוונטי (אפשר לבצע commit על קבצים בודדים / חלקים ספציפיים מתוך אותם קבצים). הפקודות המשמשות אותנו לשלב זה הן:

a. `git status`- מאפשר לבדוק את השינויים בכל הקבצים בסביבת העבודה
b. `git add` - הוספת קבצים ל-staging area. שלב זה מאפשר להתכונן ל-commit מבלי צורך לכלול את כל הקבצים שהשתנו בסביבת העבודה באותו commit.
c. `git commit` - לקיחת מאין snapshot של סביבת העבודה או במילים אחרות, יצירת commit.

3. **דחיפת השינויים ל-repo מרוחק:**

לאחר שיצרנו commits המייצגים את השינויים שביצענו, נרצה לשתף את שינויים אלו עם שאר המפתחים שאנחנו עובדים איתם. לצורך ביצוע פעולה זו נשתמש בפקודה `git push` שדוחפת את ה-commits שביצענו ב-repo המקומי ל-repo המרכזי.

4. **החלפה של ענפים:**

על מנת שכל המפתחים לא ידרכו על הרגליים אחד של השני כאשר הם מבצעים שינויים, Git מאפשרת להשתמשת בענפים. מה זה אומר?

ענף (branch) הוא פיצול שקורה בפרויקט בנקודת זמן מסוימת. פיצול זה יכול להיות לתמיד, לדוגמא, לצורך יצירת גרסה חדשה לפרויקט (צריך לתמוך בשתי גרסאות שונות). בנוסף, פיצול יכול להיות גם

זמני לצורך כתיבת קוד חדש / שינוי קוד קיים ומיזוג של ה-branch החדש עם branch הפיתוח המרכזי לאחר מכן.

ה-branch הדיפולטי ב-Git הוא ה-master וממנו אפשר לייצר branches נוספים. שתי הפקודות הבאות יהיו לנו שימושיות לצורך עבודה עם branches:

a. `git checkout -b` - יצירת branch חדש
b. `git checkout` - החלפה בין branches

5. מיזוג של ענפים:

כמו שאמרנו קודם לכן, ברוב המקרים מייצרים branch חדש לצורך שינוי קוד ב-branch הראשי. לכן, לאחר שנבצע את כל השינויים ב-branch החדש, נרצה למזג בין 2 ה-branches (או כמו שנהוג להגיד למרג'ג' - מלשון merge). הפקודה שנשתמש בה לצורת ביצוע פעולה זו, היא: `git merge`.

מאיפה הכל מתחיל?

בשביל להתחיל לעבוד עם Git, צריך להריץ את הפקודה `git init` שכל מה שהיא עושה זה לייצר repo. הפקודה הזו יוצרת מאחורי הקלעים תיקייה מוסתרת בשם `.git` המייצגת את ה-repo עצמו. התיקייה שממנה מריצים את הפקודה היא התיקייה שבה המשתמש הרגיל צריך לעבוד וכל פקודת Git אותה הוא מריץ מתורגמת לשינוי בתיקייה git.

לאורך המאמר נעבוד מתוך התיקייה git-internals - בואו נייצר בתוכה repo חדש:

```
➜  git-internals git init
Initialized empty Git repository in /home/git-internals/.git/
➜  git-internals git:(master) ✗ ls .git
branches  config  description  HEAD  hooks  info  objects  refs
```

תיקייה זו מכילה את כלל הקבצים והתיקיות הנחוצים על מנת לנהל את ה-git repo. בואו נסביר על המעניינים מביניהם:

- תיקיית objects: תיקייה המכילה את כל היסטוריית שינויי הקבצים. בתיקייה זו מתרחש כל הקסם שעליו נדבר בהמשך...
- תיקיית hooks: תיקייה המכילה סקריפטים שיכולים לרוץ לפני ואחרי פקודות Git.
- תיקיית info: תיקייה המכילה את קובץ ה-exclude המשתמש לשמירת שינויים בסביבת הפיתוח כמו הגדרות IDE וכו'. הקובץ דומה לקובץ ה-gitignore. רק שאינו נועד לשיתוף.
- תיקיית refs: תיקייה המכילה את ההגדרות של כל tag/branch עבור ה-repo. גם על תיקייה זו נעמיק בהמשך.

- קובץ ה-HEAD: קובץ המשמש את Git במעקב אחר ה-branch הנוכחי. הוא בעצם מכיל הפנייה לתיקייה refs.
- קובץ ה-config: קובץ הקונפיגורציה של ה-repo. בכל פעם שמריצים את הפקודה git config שמטרתה לשנות את קונפיגורציית ה-Git המקומית של ה-repo, קובץ הזה משתנה. יש אפשרות להריץ את פקודה זו עם דגל g- מה שיגרום לשינוי קונפיגורציה לכל ה-repos על המחשב.
- קובץ ה-description: מכיל טקסט המתאר את ה-repo.

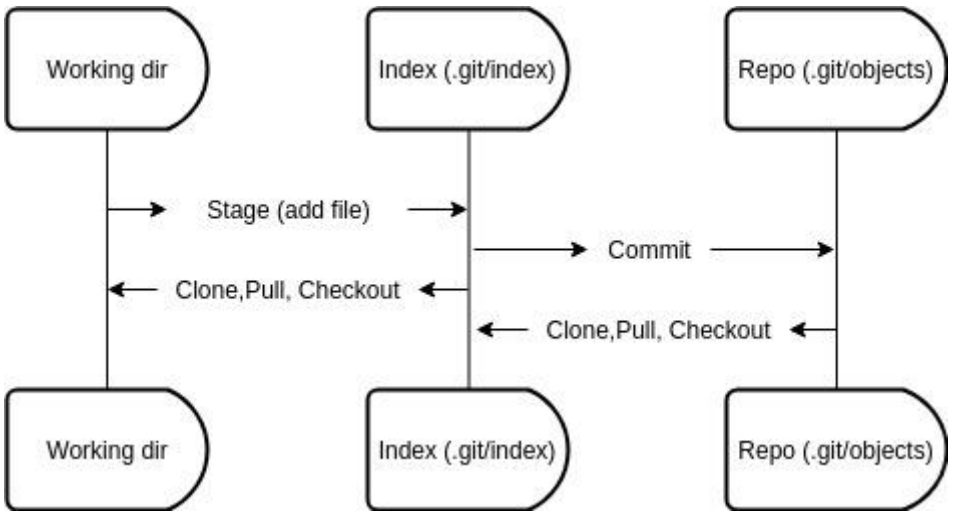
הדבר שבאמת חשוב לדעת, הוא שכאשר עושים clone ל-repo מסויים, בעצם כל מה שקורה בפועל זה העתקה של תיקיית ה-git. ממקום אחר.

כל הדרכים מובילות ל-Index

רוב פעולות ה-Git אשר מסנכרנות באופן כלשהו בין הסביבה המקומית של המשתמש לבין ה-repo (מקומי או מרוחק) עוברות בתחנת ביניים. לתחנה זו שמות שונים: staging area, cache וקובץ ה-index. אנחנו נשתמש באופציה האחרונה.

ניתן לראות בשרטוט 2 את שלושת האזורים ש-Git משתמש בהם על מנת לעבוד:

1. בצד השמאלי ביותר נמצאת מערכת הקבצים הרגילה - המקום שבו למעשה נמצאים קבצי הפרויקט (או במילים אחרות - המקום בו התבצע git clone / git init).
2. במרכז נמצא קובץ ה-index שעליו נפרט בחלק זה. מיקומו של הקובץ הוא בתיקיה .git
3. בצד הימני ביותר נמצא תוכן התיקיה .git/objects המייצג את ה-repo עצמו. על תיקייה זו נרחיב בהמשך המאמר.



[שרטוט 2: שלושת אזורי הליבה של Git]

קובץ ה-index הינו קובץ בינארי המהווה סוג של מערכת קבצים בפני עצמו, המכיל הצבעה על קבצים שונים בסביבת העבודה ושמירת metadata. מטרתו היא לעזור ל-Git לנהל את ההבדלים בין ה-working dir לבין ה-repo, ולאחר מכן, להכיל שינויים אלו.

קיימים שני כיוונים בהם ניתן לשנות את תוכן הקובץ:

- מכיוון ה-working dir אל ה-repo: כאשר המשתמש מבצע את פעולת ה-staging, או במילים אחרות, כאשר הוא מבצע את הפקודה git add. בשלב הבא, כאשר המשתמש מבצע את הפקודה git commit המידע שנשמר ב-index משמש על מנת לבצע את השינויים על ה-repo. כמובן שנרחיב שהמשך לעומק על פעולות אלו.
- מכיוון ה-repo אל ה-working dir: כאשר המשתמש מנסה להחיל שינויים מה - repo אל סביבת העבודה שלו. כמו למשל, כאשר מבצעים git pull \ clone, החלפה של branch או ביצוע merge (כאשר קיימים קונפליקטים).

שימוש בקובץ ה-index אמנם מסבך את המודל של Git, אך הוא יכול להיות מאוד שימושי כאשר רוצים לבצע פעולות ספציפיות כמו:

- staging רק על חלקים ספציפיים בקובץ, ובכך לחלק שינוי גדול באותו הקובץ להרבה commits.
- השוואה מהירה בין התוכן שלו לבין הקבצים שנמצאים ב-working dir. זה בעצם מה שקורה כאשר מבצעים git status.
- הצגת מידע על קונפליקטים בקוד המתרחשים לאחר ביצוע merge. בנוסף, כאשר יש קונפליקט של הרבה מאוד קבצים, Git מקל עלינו בכך שאפשר לבצע git add לקבצים שאין בהם / פתרנו בהם את הקונפליקט. בצורה כזאת אפשר להקל על התהליך בצורה מאוד משמעותית.

למרות שנחמד לדבר על דברים ב-high level, הרבה יותר נחמד לראות דברים בעיניים! אז בואו נתחיל לשחק קצת...

כאשר ביצענו בתחילת המאמר git init, ראינו שמתווספת לנו התיקייה .git, אך היא לא כללה קובץ index. על מנת לייצר אותו נבצע פעולות git add:

```
➜  git-internals git:(master) ✗ echo 1 > 1.txt
➜  git-internals git:(master) ✗ echo 2 > 2.txt
➜  git-internals git:(master) ✗ ls .git
branches  config  description  HEAD  hooks  info  objects  refs
➜  git-internals git:(master) ✗ git add 1.txt
➜  git-internals git:(master) ✗ git add 2.txt
➜  git-internals git:(master) ✗ ls .git
branches  config  description  HEAD  hooks  index  info  objects  refs
```

נבצע git status על מנת לראות את השינוי שנעשה:

```
➜  git-internals git:(master) ✗ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

      new file:   1.txt
      new file:   2.txt
```

נתחיל להסתכל על קובץ ה-index:

```
➜  git-internals git:(master) ✗ xxd .git/index
00000000: 4449 5243 0000 0002 0000 0002 5be7 1f8d  DIRC........[...
00000010: 15c9 8dd2 5be7 1f8d 15c9 8dd2 0001 0302  ....[...........
00000020: 01a4 07fa 0000 81a4 0000 03e8 0000 03e8  ................
00000030: 0000 0002 d004 91fd 7e5b b6fa 28c5 17a0  ........~[..(...
00000040: bb32 b8b5 0653 9d4d 0005 312e 7478 7400  .2...S.M..1.txt.
00000050: 0000 0000 5be7 1f8f 1b07 9ec0 5be7 1f8f  ....[.......[...
00000060: 1b07 9ec0 0001 0302 01a4 07fe 0000 81a4  ................
00000070: 0000 03e8 0000 03e8 0000 0002 0cfb f088  ................
00000080: 86fc a9a9 1cb7 53ec 8734 c84f cbe5 2c9f  ......S..4.O..,.
00000090: 0005 322e 7478 7400 0000 0000 04e4 0dcd  ..2.txt.........
000000a0: 213b 8eed bff1 422a f7b3 71e2 f893 b04c  !;....B*..q....L
```

Header

הקובץ מתחיל ב-header של 12 בתים (מסומן בכתום). מבנה ה-header:

- מתחיל במילה DIRC שמשמעותה היא dir cache (סתם לידע כללי - אם ישאלו אתכם פעם במונית הכסף...)
- 4 בתים של גרסת ה-Git. אני משתמש ב-2.17.1 ולכן אנחנו רואים את הערך 2.
- 4 בתים של מספר הרשומות בקובץ. עשינו `git add` לשני קבצים, ולכן רואים את הערך 2.

File entry

לאחר ה-header, ניתן לראות שקיימת רשומה לכל קובץ שהוספנו (מסומן בכחול). כל רשומה מכילה את המידע הבא (נסתכל רק על הראשונה):

- 8 בתים של ה-ctime של הקובץ (000Ch: 5B E7 1F 8D 15 C9 8D D2)

נוודא שזה הערך הנכון:

```
➜  git-internals git:(master) ✗ stat -c "%Z" 1.txt
1541873549
➜  git-internals git:(master) ✗ printf '%x\n' 1541873549
5be71f8d
```

אז אכן קיבלנו את 4 הבתים הראשונים. 4 הבתים לאחר מכן מייצגים את החלק היחסי של הזמן ב-nanosecs.

- 8 בתים של ה-mtime הקובץ (0013h: D2 5B E7 1F 8D 15 C9 8D D2)
- 4 בתים של ה-device_id של הקובץ (001Ch: 00 01 03 02)
- 4 בתים של ה-inode של הקובץ (0020h: 01 A4 07 FA)

נוודא שזה הערך הנכון:

```
➜  git-internals git:(master) ✗ ls -i 1.txt
27527162 1.txt
➜  git-internals git:(master) ✗ printf '%x\n' 27527162
1a407fa
```

- 4 בתים של הרשאות הקובץ (0024h: 00 00 81 A4)
- 4 בתים של ה-uid של המשתמש (0028h: 00 00 03 E8)

לאחר מכן, 4 בתים של ה-gid של המשתמש (002Ch: 00 00 03 E8)

```
➜  git-internals git:(master) ✗ id -u
1000
➜  git-internals git:(master) ✗ id -g
1000
➜  git-internals git:(master) ✗ printf '%x\n' 1000
3e8
```

- 4 בתים של גודל הקובץ (0030h: 00 00 00 02). גודל הקובץ במקרה שלנו הוא אכן 2 - התו '1' + ירידת שורה.
- 20 בתים של ה-Object ID של הקובץ (0034h: D0 04 91 FD 7E 5B B6 FA 28 C5 17 A0 BB 32 B8 B5 06 53 9D 4D). נפרט כיצד מחושב ערך זה בחלק הבא.
- 2 בתים של flag שמכיל state של הרשומה (0048h: 00 05)
- נתיב לקובץ (004Ah: 31 2E 74 78 74 00 : 1.txt)

על מנת לראות את כל המידע המוצג בקובץ ה-index, ניתן להשתמש בפקודה הבאה:

```
➜  git-internals git:(master) ✗ git ls-files --stage --debug
100644 d00491fd7e5bb6fa28c517a0bb32b8b506539d4d 0    1.txt
  ctime: 1541879980:354057721
  mtime: 1541879980:354057721
  dev: 66306      ino: 27527162
  uid: 1000 gid: 1000
  size: 2   flags: 0
100644 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f 0    2.txt
  ctime: 1541881219:397375528
  mtime: 1541881219:397375528
  dev: 66306      ino: 27527166
  uid: 1000 gid: 1000
  size: 2   flags: 0
```

ניתן לראות שהערכים זהים לערכים שהצגנו לפני ☺

להלן פירוט מלא של מבנה קובץ ה-index:

https://github.com/git/git/blob/master/Documentation/technical/index-format.txt

Git == DB

Git הוא בעצם מבנה נתונים לאחסון key-value. עבור כל ערך שנכניס ל-repo, נקבל מפתח תואם וייחודי שאיתו נוכל לגשת לאחר מכן לערך זה. Git משתמש בשתי אבני בניין על מנת לשמור את כל המידע: blobs ו-trees . בחלק זה, נסביר עליהן ולמה הן חשובות.

Blobs

ניתן להשתמש ב-`git hash-object` על מנת לקבל את המפתח המחושב לקובץ מסוים ובנוסף ליצור אובייקט חדש ב-repo:

```
➜  git-internals git:(master) ✗ echo asdf > asdf.txt
➜  git-internals git:(master) ✗ git hash-object -w asdf.txt
8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a
```

קיבלנו SHA-1 של האובייקט שיצרנו הנקרא גם **blob**. כעת, נשתמש בפקודה `git cat-file` המספקת מידע כמו תוכן / גודל / סוג על אובייקטים בתוך ה-repo:

```
➜  git-internals git:(master) ✗ git cat-file -p
8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a
asdf
```

נחמד מאוד, הכנסנו ערך וקיבלנו חזרה. באמת כל הכבוד Git... מאוד התרשמנו!

אז איפה בעצם נשמר כל המידע? כמו שהבטחתי לכם מקודם, הגיע הזמן לדבר על התיקייה .git/objects בואו נציץ בתוכה:

```
➜  git-internals git:(master) ✗ ls .git/objects
0c  8b  d0  info  pack
```

מעניין... תיקייה בשם 8b אולי קשורה ל-SHA שלנו שמתחיל באותן אותיות? בואו נכנס אליה:

```
➜  git-internals git:(master) ✗ cd .git/objects/8b
➜  8b git:(master) ls
d6648ed130ac9ece0f89cd9a8fbbfd2608427a
➜  8b git:(master) file d6648ed130ac9ece0f89cd9a8fbbfd2608427a
d6648ed130ac9ece0f89cd9a8fbbfd2608427a: zlib compressed data
```

אז מה קיבלנו?

8b + d6648ed130ac9ece0f89cd9a8fbbfd2608427a ==

8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a

Git בעצם משתמש ב-2 האותיות הראשונות על מנת לסדר את כל האובייקטים. עבור כל אובייקט נשמר קובץ zlib... ננסה לפתוח אותו:

```
➜  8b git:(master) zlib-flate -uncompress <
d6648ed130ac9ece0f89cd9a8fbbfd2608427a
blob 5asdf
```

ניתן לראות ש-Git מחשב לכל blob ב-repo את ערך ה-160bit SHA1 על הפורמט הבא:

```
blob{space}{file-length in bytes}{null-termination} {file-data}
```

והנה עוד הוכחה - נחשב SHA1 על הפורמט ש-Git מחשב, ונראה שנקבל את אותו הערך:

```
➜  8b git:(master) printf "blob 5\0asdf\n" | sha1sum
8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a  -
```

Trees

כמו שניתן להסיק, Git משתמש ב-blobs על מנת לשמור את המצב של קבצים בודדים. אבל זה לא מספיק, נרצה גם לשמור על קשר בין הקבצים הללו, וקשר בין ה-blob שמייצג את התוכן לבין הנתיב שלהם. את התפקיד הזה בדיוק באים לבצע העצים. כאמור, עץ הינו עוד אובייקט ששומר על תוכן עם פורמט של רשימת blobs בצורה:

```
{file-mode} {object-type} {object-hash}\t{file-name}\n
```

שדה ה-file-mode משתמש לשמירת ההרשאות של כל אובייקט בעץ. כאשר Git מעתיק את הקבצים ל-working dir, הוא צריך לשמור על ההרשאות המקוריות שלהן, ולכן המידע הזה נשמר בזמן יצירת העץ. להלן הערכים האפשריים לשדה זה:

- 040000 (0100000000000000): תיקייה
- 100644 (1000000110100100): קובץ עם הרשאות קריאה בלבד
- 100664 (1000000110110100): קובץ עם הרשאות כתיבה וקריאה בלבד
- 100755 (1000000111101101): קובץ הרצה
- 120000 (1010000000000000): קובץ Symbolic link
- 160000 (1110000000000000): קישור ברמת Git

ננסה לבנות עץ משלנו - לצורך כך, נשתמש ב-3 הקבצים שייצרנו עד עכשיו ונחשב את פורמט הקלט לצורך יצירת עץ:

```
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d 1.txt
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f 2.txt
100644 blob 8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a asdf.txt
```

נשמור את התוכן בקובץ בשם tree.txt ונשתמש בפקודה git mktree על מנת לייצר אובייקט עץ:

```
➜  git-internals git:(master) ✗ git mktree < tree.txt
d309906a8355ad79041ca838b40dadddc9fae387
➜  git-internals git:(master) ✗ ls .git/objects
0c  8b  d0  d3  info  pack
➜  git-internals git:(master) ✗ git cat-file -p d30990
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d 1.txt
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f 2.txt
100644 blob 8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a asdf.txt
```

ניתן לראות שבדומה לפקודה git hash-object, גם הפקודה git mktree יוצרת אובייקט חדש תחת תיקיית ה-objects.

מלבד אוסף של blobs, כל עץ יכול להצביע על עצים אחרים - במילים אחרות, עץ משמש את **Git** לייצוג תיקייה. בצורה כזאת, Git מייצג את מבנה התיקיות והקבצים על ידי פורמט בסיסי של עץ. על מנת לוודא שאנחנו מבינים, ניצור עוד תיקייה (בתוך התיקייה הנוכחית שלנו), ונעתיק את 1.txt לתוכה:

```
➜  git-internals git:(master) ✗ mkdir internal-dir
➜  git-internals git:(master) ✗ cp 1.txt internal-dir
```

בשל העובדה שלא שינינו את תוכן הקובץ, ה-SHA שלו יהיה בדיוק אותו הדבר. תוכן אובייקט העץ של התיקייה internal-dir יהיה:

```
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d 1.txt
```

כעת נוכל לגרום לעץ הקודם להצביע על העץ החדש שלנו, וליצר אובייקט עץ חדש עם התוכן:

```
100644 blob d00491fd7e5bb6fa28c517a0bb32b8b506539d4d 1.txt
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f 2.txt
100644 blob 8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a asdf.txt
040000 tree 6d2b647a0bb32c9e648ed130afbbfd2608427a23 internal-dir
```

אוקי... אז עכשיו בוא נבין איך הכל מתחבר...

עד עכשיו כיסינו את אבני הבניין הבסיסיות שבהן Git משתמש. בחלק זה ננסה לחבר את כל החלקים ולהסביר כיצד הפקודות שאנו מריצים, עובדות מאחורי הקלעים.

git add

אז אני סומך עליכם שאת פקודה זאת אתם כבר יודעים להסביר... אבל ארשום את ההסבר המלא רק כדי לוודא שאני מבין ☺

כאשר אנחנו מריצים את פקודה זו, מתרחשים 2 דברים:

- נוצרים אובייקטים חדשים לתיקיית ה-objects המייצגים את התוכן שהוספנו - blobs לקבצים ועצים לתיקיות
- מידע המצביע על אובייקטים אלו נכתב לקובץ ה-index

כאשר אנחנו לא מוסיפים קבצים חדשים לגמרי, אלא רק משנים את התוכן של קבצים שכבר הוספנו לפני, עדיין מבצעים את שני השלבים האלו. כאמור, ל-Git לא אכפת מה השינוי שעשינו לקובץ - הוא מחשב את ה-SHA רק על סמך תוכן הקובץ. השינוי היחיד, הוא שכעת לא נוסיף שורה חדשה לקובץ ה-index, אלא נשנה את השורה הקיימת שמייצגת את הקובץ ששינינו.

git commit

בעבודה שוטפת של Git אף אחד לא מתעניין ב-blobs או בעצים - רק ב-commits.

אז מה זה בעצם commit אתם שואלים?

אם עקבתם עד עכשיו, התשובה היא ממש פשוטה - **commit** הוא בעצם עטיפה מעל עצים!

בעת הרצת הפקודה, Git בונה אובייקט עץ חדש מהמידע שנמצא בקובץ ה-index ושומר את האובייקט לתיקיית ה-objects. בואו נעשה את זה:

```
➜  git-internals git:(master) ✗ git commit -m "1st commit"
[master (root-commit) efb845c] 1st commit
 3 files changed, 3 insertions(+)
 create mode 100644 1.txt
 create mode 100644 2.txt
 create mode 100644 asdf.txt
➜  git-internals git:(master) ✗ ls .git/objects/
0c  8b  d0  d3  ef  info  pack
➜  git-internals git:(master) ✗ git cat-file -p efb845c
tree d309906a8355ad79041ca838b40dadddc9fae387
```

```
author Or Kamara <orkamara@gmail.com> 1542473681 +0200
committer Or Kamara <orkamara@gmail.com> 1542473681 +0200
gpgsig -----BEGIN PGP SIGNATURE-----
 1234
 -----END PGP SIGNATURE-----
1st commit
```

ניתן לראות שנוצר אובייקט חדש מסוג commit. כל אובייקט כזה מכיל:

- את ה-ID של אובייקט העץ (שאותו ראינו מקודם)
- מידע על מי שאחראי על ה-commit
- חתימת GPG על ה-commit (אופציונאלי). נרחיב על כך בהמשך.
- commit message

ה-SHA שחושב לאובייקט ה-commit הוא בעצם ה-commit ID שאנחנו מכירים!

```
➜  git-internals git:(master) ✗ git log
commit efb845cf53e6724ab2c48b0354eabd9451d70960 (HEAD -> master)
Author: Or Kamara <orkamara@gmail.com>
Date:   Sat Nov 17 18:54:41 2018 +0200
1st commit
```

כדי ליצור את אובייקט ה-commit, נעשה שימוש מאחורי הקלעים בפקודה `git commit-tree`. הפקודה מקבלת את ה-id של אובייקט העץ (של התיקייה הראשית של הפרויקט) ופרמטרים נוספים כמו ה-commit message וממנו מייצרת אובייקט חדש המייצג את ה-commit:

```
➜  git-internals git:(master) ✗ git commit-tree d309906 -m
"digitalwhisper 101"
e6478017f365ec5bb017012142e1a59a7a65d229
➜  git-internals git:(master) ✗ git cat-file -p e64780
tree d309906a8355ad79041ca838b40dadddc9fae387
author Or Kamara <orkamara@gmail.com> 1542959225 +0200
committer Or Kamara <orkamara@gmail.com> 1542959225 +0200

digitalwhisper 101
```

כמובן שנוצר לנו אובייקט commit חדש המכיל את אותו העץ, אך עם commit message אחר.

git log

פקודה זו מציגה לנו את היסטוריית ה-repo, או במילים אחרות את הקשר בין כל ה-commit objects.

בשביל להבין את זה, נייצר commit נוסף:

```
➜  git-internals git:(master) ✗ echo 11 > 1.txt
➜  git-internals git:(master) ✗ git add 1.txt
➜  git-internals git:(master) ✗ git commit -m "2nd commit"
[master e497d89] 2nd commit
 1 file changed, 1 insertion(+), 1 deletion(-)
➜  git-internals git:(master) ✗ ls .git/objects
0c  81  8b  b4  d0  d3  e4  ef  info  pack
➜  git-internals git:(master) ✗ git cat-file -p e497d89
tree 81f45b0feb284aac3c1bc725843d288375d1b238
parent efb845cf53e6724ab2c48b0354eabd9451d70960
author Or Kamara <orkamara@gmail.com> 1542474835 +0200
committer Or Kamara <orkamara@gmail.com> 1542474835 +0200
gpgsig -----BEGIN PGP SIGNATURE-----
1234
 -----END PGP SIGNATURE-----
2nd commit
➜  git-internals git:(master) ✗ git cat-file -p 81f45b
100644 blob b4de3947675361a7770d29b8982c407b0ec6b2a0 1.txt
100644 blob 0cfbf08886fca9a91cb753ec8734c84fcbe52c9f 2.txt
100644 blob 8bd6648ed130ac9ece0f89cd9a8fbbfd2608427a asdf.txt
➜  git-internals git:(master) ✗ git cat-file -p b4de39
11
```

לאחר שיצרנו את ה-commit החדש, נוצר לו אובייקט תואם (e497d89). העץ שעליו מצביע commit זה (81f45b) מצביע על אותם הקבצים, אך על blob חדש (b4de39) עבור הקובץ אותו שינינו. ההבדל מאובייקט ה-commit הקודם הוא השדה parent, אשר מצביע על אובייקט ה-commit הקודם (efb845).

כלומר, בעזרת שרשרת של commits, ניתן לייצר את ההיסטוריה:

```
➜  git-internals git:(master) ✗ glog
* e497d89 (HEAD -> master) 2nd commit
* efb845c 1st commit
```

איך **Git** יודע איפה אנחנו עכשיו?

ראינו איך היסטוריית ה-commits עובדת מאחורי הקלעים - אך כאמור, חוץ מאשר להסתכל על ההיסטוריה, אפשר לחזור בזמן ל-commit מסויים בעזרת הפקודה git checkout. כמו שאתם יכולים לדמיין, פקודה זו מתבצעת על ידי החלפת אובייקט ה-commit ותוכן ה-index. אז איך Git יודע על איזה אובייקט commit הוא מצביע עכשיו? הוא משתמש בקובץ git/HEAD.:

```
➜  git-internals git:(master) ✗ git checkout efb845c
Previous HEAD position was e497d89 2nd commit
HEAD is now at efb845c 1st commit
➜  git-internals git:(efb845c) ✗ cat 1.txt
1
➜  git-internals git:(efb845c) ✗ cat .git/HEAD
efb845cf53e6724ab2c48b0354eabd9451d70960
➜  git-internals git:(efb845c) ✗ git checkout e497d89
Previous HEAD position was efb845c 1st commit
HEAD is now at e497d89 2nd commit
➜  git-internals git:(e497d89) ✗ cat 1.txt
11
➜  git-internals git:(e497d89) ✗ cat .git/HEAD
e497d89542590272c3cd38b7acbb713b26806775
```

בדוגמא זו ראינו כיצד git checkout מחליף את תוכן הקבצים בתיקייה המקומית, וגם את תוכן הקובץ git/HEAD. שיצביע על ה-commit הנוכחי.

שימו לב שבדוגמא ביצענו checkout על commit ולא על branch. בדוגמא הבאה, נבצע את האופציה השנייה על מנת להבין את ההבדל:

```
➜  git-internals git:(master) ✗ git checkout master
Switched to branch 'master'
➜  git-internals git:(master) ✗ cat .git/HEAD
ref: refs/heads/master
➜  git-internals git:(master) ✗ cat .git/refs/heads/master
e497d89542590272c3cd38b7acbb713b26806775
```

בעצם שינינו את ה-branch ל-master (האופציה היחידה כרגע), שכאמור מצביע על אותו commit בדיוק (e497d89). מאחורי הקלעים ניתן לראות שהערך הכתוב בקובץ ה-HEAD משתנה מערכו של ה-commit לנתיב של קובץ התואם ל-master branch. תוכנו של הקובץ בנתיב זה הוא (תאמינו או לא) בדיוק אותו ה-commit שהיה מקודם (e497d89).

git branch

באותו האופן שבו ה-master הצביע על ה-commit המתאים, בכל יצירה של branch חדש נוצר קובץ טקסט שתוכנו הוא ה-SHA על אובייקט ה-commit התואם לו:

```
➜  git-internals git:(master) ✗ git checkout -b "branch1"
Switched to a new branch 'branch1'
➜  git-internals git:(branch1) ✗ cat .git/HEAD
ref: refs/heads/branch1
➜  git-internals git:(branch1) ✗ ls .git/refs/heads
branch1  master
➜  git-internals git:(branch1) ✗ cat .git/refs/heads/branch1
e497d89542590272c3cd38b7acbb713b26806775
```

בשל העובדה ש-branch1 יצא מתוך ה-master, הם מצביעים לאותו אובייקט commit.

Commits spoofing

חלק זה של המאמר אמנם סוטה מהנושא העיקרי, אך מטרתו לחשוף אתכם לקונספט אבטחה בעייתי שאפשר להימנע ממנו בקלות.

אז מה הבעיה? תארו לכם שאתם עובדים בחברת ענק כחלק מפרויקט עצום (או סתם על פרוייקט open-source) שעליו עובדים עשרות מפתחים. כל מפתח אחראי על ה-commits שלו, ואם יש באג או בעיה בקוד, אפשר לדעת מה ה-commit שבו זה קרה, ולהגיע אל המפתח המתאים. רגע, בעצם... אם היה אפשר לייצר commits בשם מפתחים אחרים, מה מונע ממפתח מסויים להכניס באגים מכוונים (כמו Backdoor) בשם מישהו אחר?

כאשר מתאמתים מול GitHub (או כל Git server אחר) עם מפתח SSH, האימות לא הופך להיות חלק מה-repo, מה שאומר ש-Github מקבל commit ומנסה לשייך אותו לאדם שרשום בתוך אובייקט ה-commit. כלומר, אם משנים את ה-commit כך שיכיל פרטים של משתמש אחר...בעיות! לצורך הדגמה נייצר repo חדש ב-GitHub עם commit ראשוני, ונבצע ממנו clone:

```
temp git clone git@github.com:orkamara/GitInternals.git
Cloning into 'GitInternals'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
➜  GitInternals git:(master) ✗ git log
commit 12d6291993116ef3ee39b80b9f4f4bc9c091e57b
Author: Or Kamara <orkamara@gmail.com>
Date:   Mon Nov 26 07:03:26 2018 +0200
    Initial commit
```

ניתן לראות שה-author של ה-commit הוא עבדכם הנאמן. כעת, נשנה את הקונפיגורציה:

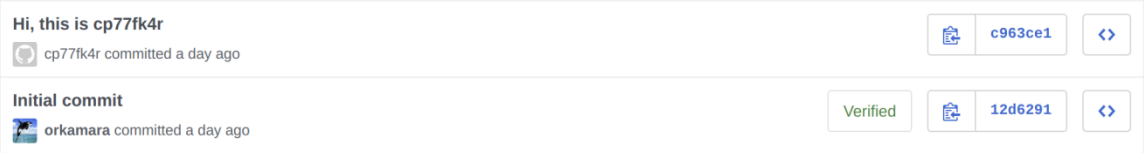
```
➜  GitInternals git:(master) ✗ git config --global user.email
"empty0page@gmail.com"
➜  GitInternals git:(master) ✗ git config --global user.name
cp77fk4r
```

לאחר מכן, נבצע commit נוסף:

```
➜  GitInternals git:(master) ✗ echo $'\nNew line' >> README.md
➜  GitInternals git:(master) ✗ git add .
➜  GitInternals git:(master) ✗ git commit -m "Hi, this is cp77fk4r"
[master c963ce1] Hi, this is cp77fk4r
```

```
 1 file changed, 2 insertions(+), 1 deletion(-)
➜  GitInternals git:(master) ✗ git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 271 bytes | 271.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:orkamara/GitInternals.git
   12d6291..c963ce1  master -> master
```

הנה התוצאה ב-GitHub:



נוצר commit חדש עם שם המשתמש **cp77fk4r**. ל-GitHub לא באמת אכפת מי ביצע את ה-commit והיא מציג את המשתמש בצורה תקינה. כעת, נחזיר את ההגדרות למשתמש **orkamara** ונייצר commit נוסף:

```
➜  GitInternals git:(master) ✗ git config --global user.name orkamara
➜  GitInternals git:(master) ✗ git config --global user.email
orkamara@gmail.com
➜  GitInternals git:(master) ✗ echo $'\nOriginal line' >> README.md
➜  GitInternals git:(master) ✗ git add .
➜  GitInternals git:(master) ✗ git commit -m "Hi, this is orka - the
commit is not signed"
[master 1854cd0] Hi, this is orka - the commit is not signed
 1 file changed, 2 insertions(+)
➜  GitInternals git:(master) ✗ git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 290 bytes | 290.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:orkamara/GitInternals.git
   c963ce1..1854cd0  master -> master
```

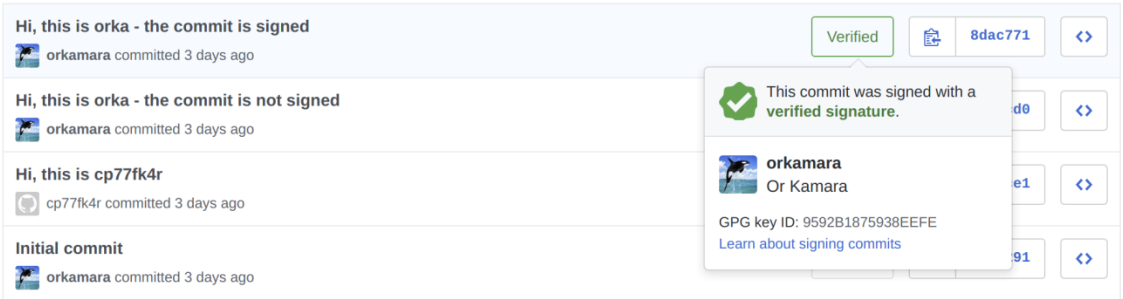
להלן ה-commit החדש שנוצר ב-GitHub - עם שם המשתמש: **orkamara**

מה אפשר לעשות על מנת לוודא ש-**commit** בטוח? נשתמש בפיצ'ר של Git לחתום commits ובמנגנון מובנה ב-GitHub. חתימת commits בעצם מאפשרת לוודא שמי שנטען שכתב את הקוד, אכן כתב אותו. Git משתמש בהצפנת GPG על מנת לוודא ש-commit שנחתם מקומית במחשב של משתמש, מצליח להתאמת מול מפתח פומבי שהמשתמש הוסיף לחשבון ה-GitHub שלו לפני. בכל פעם שמייצרים commit חתום, מתווסף לאובייקט ה-commit שדה נוסף המכיל את מפתח ה-GPG הפומבי.

GitHub משתמש במידע זה ובמפתח פומבי שמכניסים להגדרות ה-repo על מנת לוודא שה-commit חוקי. נאפשר ל-Git החתום commits ונייצר אחד חדש:

```
➜  GitInternals git:(master) ✗ git config --global commit.gpgsign
true
➜  GitInternals git:(master) ✗ echo 1 > README.md
➜  GitInternals git:(master) ✗ git add .
➜  GitInternals git:(master) ✗ git commit -m "Hi, this is orka - the
commit is signed"
[master 8dac771] Hi, this is orka - the commit is signed
 1 file changed, 1 insertion(+), 4 deletions(-)
➜  GitInternals git:(master) ✗ git push
Counting objects: 3, done.
Writing objects: 100% (3/3), 920 bytes | 920.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:orkamara/GitInternals.git
   1854cd0..8dac771  master -> master
```

כעת GitHub מזהה שה-commit חתום ואפילו מוסיף לנו סימן של verified עליו:



לא נסביר במסגרת המאמר מה בדיוק צריך להגדיר בסביבה לקבלת פיצ'ר זה, ולכן אפנה לקריאה יותר מעמיקה: https://help.github.com/articles/about-commit-signature-verification/

סיכום

במאמר זה נכנסנו למאחורי הקלעים של Git במטרה להבין איך הפעולות הבסיסיות שאנחנו משתמשים בהן כחלק מתהליך הפיתוח עובדות. למדנו על קובץ ה-Index, מה הן מטרותיו, מה הפורמט שלו ואיך הוא משרת את Git. לאחר מכן למדנו על אבני הבניין של Git - עצים ו-blobs, על התוכן שלהן ועל השימוש בהן. לאחר מכן המשכנו לדבר בפירוט על כל הפקודות הבסיסיות של Git - כיצד הן עובדות ואיך הם משנות את תוכן התיקייה git. קינחנו עם הצגת signed commits, למה הם חשובים ואיך משתמשים בהם.

אין ספק שאפשר להיות אשף ב-Git גם מבלי להבין איך דברים עובדים מתחת לפני השטח, אך אני בטוח כי הידע שתיקחו איתכם ממאמר זה יעזור לכם לשחק עם Git יותר בביטחון ולהשתמש בקונספטים דומים כאשר תפתחו את ה-Source control הבא שלכם!

על המחבר

אור קמארה - בן 29, ראש צוות פיתוח בחברת Snyk, בעל ניסיון במחקר ופיתוח של מוצרי אבטחה, עם דגש על Windows internals ורשתות.

לשאלות, התייעצויות והערות:

Email: OrKamara@gmail.com

Linkedin: https://www.linkedin.com/in/or-kamara-009a0bb8/

Github: https://github.com/orkamara