לימוד עצמי 1 – חזרה על גיט

עבר הרבה זמן מאז שהתראינו, הא? סיימנו את סמסטר א', הייתה לנו אחלה של חופשה, והגיע הזמן לחזור וללמוד חומר חדש ומרגש! יאיי! אבל רגע לפני שאנחנו מתחילים עם חומר חדש, צריך לחזור על החומר שכבר למדנו, ואולי לחדד כמה נושאים שלא כל כך הבנו בסמסטר הקודם.

בהתחלה נדבר קצת תיאוריה, ואחר כך נדבר על פקודות של גיט בפועל.

מה זה מערכת ניהול גרסאות?

בואו נעשה תרגיל בדמיון מודרך:
דמיינו שאתם משחקים במשחק המחשב Minecraft. (מכירים, נכון?)
פתחתם עולם חדש ונקי, והחלטתם שבא לכם לבנות בית קטן ונחמד.
עבדתם כמה שעות, השקעתם ממש ויצא לכם בית יפה ונחמד.
וואלה באמת יצא אחלה של בית. עכשיו בא לכם להוסיף עוד קומה לבית, אבל זה קצת מפחיד... כי אם משהו ייהרס או שסתם ייצא מכוער - אי אפשר יהיה לחזור אחורה למצב הנוכחי שממש יצא נחמד ויפה.
אז שמרת את המצב הנוכחי של המשחק, והתחלת לבנות עוד קומה.
פתאום חבר שלך הצטרף אליך למשחק, ורוצה לבנות גינה מסביב לבית שלך.
אבל שניכם מאוד תפריעו אחד לשני, ואפשר לעשות את העבודה שלכם במקביל, אז חבר שלך לוקח את הגרסה האחרונה ששמרת, מעתיק אותה אליו למחשב, ומתחיל לפתח בה את הגינה הממש-יפה-ומושקעת שלו.
אחרי כמה שעות אתם רוצים לאחד את השינויים שעשיתם, אז אתם מעלים את הגרסאות לשרת מרכזי, וממזגים את העבודה שכל אחד עשה על המחשב שלו בהעתק המקומי שלו, לתוצר אחד שלם.

זהו, סיימנו עם הדמיון המודרך. בואו נדבר על קוד.

כשאנחנו עובדים על פרויקט קוד, חשוב לנו לשמור את העבודה שלנו. חשוב לנו שיהיה אפשר לחזור אחורה אם פיקשנו משהו במהלך העבודה. חשוב לנו שתהיה אפשרות לעבוד ביחד עם אנשים נוספים על הפרויקט שלנו. חשוב לנו שתהיה דרך יעילה ונוחה לשמור את השינויים שעשינו, ולעבוד במקביל על הפרויקט מכמה מחשבים שונים.

בדיוק בשביל זה פיתחו **מערכת לניהול גרסאות**. בעצם מדובר בתוכנה שיודעת לעזור לנו לשמור מדיי פעם את השינויים שביצענו, כדי שנוכל לחזור אחורה בעתיד אם נצטרך. היא גם מאפשרת לנו לעבוד ביחד על הפרויקט, ואם יש באג בקוד – לדעת מי כתב אותו (כדי שנוכל להרביץ לו, כמובן).

יש המון דרכים לשמור את השינויים שלנו. אפשר להעביר קבצים לחבר שלי בוצאפ כדי שיהיו לו השינויים שעשיתי, ואפשר גם כל כמה שעות להעתיק את כל הקוד שלי הצידה ולעשות לו גיבוי. אפשר לעשות את זה, אבל זו דרך נוראית לעבודה. היא ממש לא נוחה, ומה אם עובדים על הפרויקט 20 אנשים?

אהה נכון, אפשר אולי במקום זה להשתמש ב-**Git!**

וואלה מגניב, אבל מה זה Git?

בקצרה – זו מערכת ניהול הגרסאות הכי נפוצה שיש כיום, והיא גם מאוד נוחה, יחסית קלה ופשוטה, אך מכילה גם המון פיצ'רים שאפשר לעשות איתם דברים מתקדמים.

גיט פותחה ע"י Linus Torvalds – המפתח של מערכת ההפעלה לינוקס. הוא חיפש דרך טובה לעבוד על פרויקט הקוד-הפתוח שלו ביחד עם אנשים נוספים, ובשביל זה הוא פיתח את Git. בחור מוכשר.

משתמשים ב-Git כמעט בכל פרויקט פיתוח קוד. שימוש בגיט זו מיומנות בסיסית שחייבת להיות לכל מתכנת. אי אפשר לעבוד על פרויקט גדול (אפילו כזה שעובדים עליו 2 אנשים) בלי עזרה של גיט. לכן בתור חניכים בתוכנית מגשימים, חשוב שנלמד לעבוד בצורה נכונה עם גיט ונדע להשתמש בו בצורה טובה.

טוב אז השתכנענו למה Git זה חשוב, עכשיו בואו נראה איך משתמשים בו.

מושגים בסיסיים

הכול התחיל בבית של חבר שלי לפני שבוע. דיברנו וזה, ופתאום עלה לנו רעיון פצצה לתוכנה סופר מגניבה. פתחנו את המחשב שלי (כמובן ההוא שקיבלתי ממגשימים) ופתחנו תיקייה חדשה לפרויקט. הוספנו קובץ בשם my-working-code.cpp, והוספנו בו קצת קוד. אחרי כמה שעות הצלחנו לגרום לו לעבוד.

עכשיו צריך לשמור את השינויים. אז יצרנו Repository של git בתיקייה שלנו. אפשר לראות ב-repo מעין 'משטח עבודה' של גיט. במשטח העבודה שלו הוא יודע לעקוב אחר שינויים בקבצים, ולהתייחס לכל משטח העבודה כאל פרויקט אחד.

פשוט הרצנו את הפקודה `git init` בתיקייה, ונוצר לנו repo חדש.

אם נריץ `git status` בתיקייה, נראה את הפלט הבא:

```
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        my-working-code.cpp
```

הקובץ שלנו כרגע נמצא במצב של Untracked File. אם נבצע שינוי בקובץ – גיט לא יעקוב אחריו, בגלל שלא הוספנו את הקובץ למעקב.

נריץ `git add my-working-code.cpp` כדי להוסיף את הקובץ למעקב. כעת הקובץ נמצא בשלב שנקרא **Staging Area** – שזה אומר: גיט שמר את המצב הנוכחי, אבל רק באופן זמני. אני יכול פשוט לבטל את השמירה הזו של השינויים בלי בעיה. אם לדוגמה אני משנה את התוכן של הקובץ, המצב החדש יהיה:

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

        new file:   my-working-code.cpp

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes...

        modified:   my-working-code.cpp
```

אפשר לראות שה**שינוי הקודם** שלנו (זה שעשינו לו add) נשמר, וה**שינוי החדש** שלנו עדיין לא נשמר. אם נריץ `git add my-working-code.cpp` שוב, השינוי החדש יישמר גם, ונגיע למצב הבא:

```
No commits yet

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

        new file:   my-working-code.cpp
```

השינויים שלנו שמורים באופן זמני. אחרי שבדקנו שוב את הקוד וראינו שהוא עובד יופי, אנחנו רוצים 'להתחייב' לשינוי החדש. בעצם אנחנו אומרים: "זו גרסה טובה של הקוד שלי. אני רוצה לשמור אותה!".

כדי להתחייב לשינוי, נריץ `git commit -m "First working code"` והופ! שמרנו. שימו לב שהוספנו לפקודה commit את הדגל **-m** שמאפשר לנו לרשום הודעה קצרה שמתארת את מה ששינינו ב-commit הזה.

נשתדל לכתוב תיאור מובן, כדי שאחר כך נוכל להבין מה שינינו בכל commit. כמובן שאין צורך לעשות commit בכל פעם שאנחנו משנים כמה שורות בקובץ! צריך להפעיל שיקול דעת ולחשוב – "האם הגעתי למצב בקוד שהייתי רוצה לשמור בצד?".

בנוסף, נשתדל גם לא לבצע כמות גדולה של שינויים ב-commit אחד. כלל האצבע הוא – אם המילה "and" מופיעה בתיאור שלכם – סימן שאפשר לחלק את זה ל-2 commit-ים נפרדים.

Log, Log, ולא שוכח

כדי לראות את ההיסטוריה של השינויים שעשינו, נריץ `git log` :

```
commit 5daee4296381c7d17a9fdf376e63a53f7927ea34 (HEAD -> master)
Author: Madrich <madrich@magshimim.org>
Date:   Sun Jan 1 00:00:00 2020 +0200

    First working code
```

כאן אנחנו רואים את הגרסה האחרונה ששמרנו (אם נוסיף commit-ים נוספים, הם יופיעו גם כאן). ה-commit העליון ביותר הוא העדכני ביותר. אפשר לראות גם את השעה שבה זה נשמר, ומי ביצע את השינוי. בנוסף, אפשר לראות את ה**מזהה הייחודי** של ה-commit. זה מה שמבדיל בינו לבין השאר. המזהה מיוצר בעזרת hash על תוכן הקובץ.

חשוב לא להתבלבל עם התיאור של ה-commit – הוא לא מזהה ייחודי, משום שיכולים להיות כמה commit-ים עם אותו תיאור.

המשכנו לפתח את הפרויקט שלנו, ואחרי שעה משהו נהרס והקוד לא מתקמפל. איזה באסה! טוב, מזל ששמרנו גרסה שעבדה! אבל רגע -

איך חוזרים אחורה?

תלוי כמה אחורה רוצים לחזור. יש מספר דרכים להחזיר אחורה שינויים שביצענו. כדי לדעת מה המצב שלנו, נסתכל על `git status` . יש מספר מצבים של שינוי:

1 - שינויים בשלב ה-Untracked File

קובץ שיצרנו בתקייה אך לא הוספנו למעקב של גיט

```
Untracked files:
  (use "git add <file>..." to include in what will be c

        not-working-code.cpp
```

כדי למחוק את השינויים האלה, נשתמש בפקודה `git clean --force` - הפקודה מסירה את כל השינויים שלא עשינו להם add.

2 - שינויים שעשינו להם `git add`

או בשמם – קבצים שנמצאים ב-Staging Area.
שינויים שעוד לא עשינו להם commit.

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   not-working-code.cpp
```

הדרך למחוק את השינויים היא עם הפקודה `git reset` . אם נריץ אותה כך, כל השינויים שנמצאים ב-Staging Area ייעלמו, ויחזרו למצב שהיה לפני שעשינו add (מצב Untracked), מה שאומר שהשינויים עדיין יהיו בתיקייה שלנו!

כדי למחוק את השינויים לגמרי, נעשה `git reset --hard` , וכך השינויים יימחקו גם מהתיקייה.

3 – שינויים שנשמרו ב-commit

שמרתי את השינויים שלי עם commit, וגיליתי שיש בהם באג.

את השינויים האלה לא נראה ב-status, אלא ב-log. ה-status שלנו אמור להיות: " nothing to commit, working tree clean". נראה את ה-log, ונמצא את המזהה של ה-commit שאליו אנחנו רוצים לחזור (מספיק להעתיק את בערך 10~ התווים הראשונים, לא חייב את כל ה-hash הארוך) ונשתמש בו.

חשוב לדעת שיש כמה דרכים לחזור אחורה ל-commit:

- עם הפקודה `git revert <commit-hash>`
- עם הפקודה `git checkout <commit-hash>`
- עם הפקודה `git reset <commit-hash>`

כל אחת מהאפשרויות פועלת בצורה שונה, ותוכלו לקרוא עליהן עוד באינטרנט / בעזרת הוספת הדגל h– או help– לפקודה.

טיפ: אם תרצו לחזור אחורה בצורה מהירה ונקייה (השינויים יימחקו גם מהתיקייה) – האופציה הטובה ביותר כנראה תהיה `git reset --hard <commit-hash>` אך **זו גם אפשרות דיי מסוכנת**, כי ה-commit-ים שעשינו נעלמים מהלוג. **השתמשו בה בזהירות.**

איך מעלים פרויקט לשרת מרוחק?

עד עכשיו דיברנו על השלבים בעבודה עם גיט **בצורה לוקאלית**. דיברנו על השלבים השונים שבהם קובץ יכול להיות, ואיך לשמור שינויים או למחוק אותם. אבל לעבוד על פרויקט שנמצא רק על המחשב שלי זה נחמד, אבל גם מסוכן. מה אם פתאום תהיה שריפה והמחשב יישרף? מה אם נרצה לעבוד עם אנשים נוספים על הפרויקט?

בדיוק בגלל זה, גיט בנוי בצורה כזו שמאפשרת להעלות את הקוד שלנו לשרת מרוחק (בשפה המקצועית – Remote Repo).

לפני שנתחיל לעבוד על פרויקט, ניצור Repo חדש בשרת. בקורס עקרונות משתמשים בשרת של חברת GitLab (בגלל שהוא מאוד נוח ויש בו פיצ'רים טובים). נתחבר למשתמש שלנו, וניצור פרויקט (Repository) חדש.

אחרי שיצרנו repo על השרת (נקרא לו מעכשיו Remote Repo), נרצה "לְשַׁבֵּט" אותו למחשב שלנו. לשם כך נריץ `git clone <url-to-project.git>` בתוך התיקייה שאליה נרצה שהפרויקט יועתק. מצאו בדף של הפרויקט שלכם באתר את הכתובת של הפרויקט. לרוב, כתובת כזו נגמרת ב-".git".

מגניב! עכשיו יש לנו העתק מקומי של ה-Remote Repo (והוא נקרא Local Repo).

התחלנו לפתח את הפרויקט על המחשב, הרצנו אותו, ומדיי פעם גם שמרנו commit-ים בשלבים שבהם הגענו לגרסה טובה שעובדת. סיימנו לעבוד להיום, ונשאר לנו רק -

להעלות את הקוד לשרת

אחרי שיש לנו כמה commit-ים ב-local repo שלנו, אנחנו רוצים "לדחוף" אותם לשרת. **שימו לב:** רק ה-commit-ים שעשיתי יועלו לשרת. שינויים שלא עשיתי להם commit עדיין, לא יועלו!

כדי "לדחוף" אותם לשרת, נריץ את הפקודה `git push origin master` . כרגע לא נתעכב על מה זה origin או master, בגלל שנלמד את זה בסדנת הלמידה הבאה בהרחבה.

ניכנס לפרויקט שלנו באתר, ונראה שהשינויים שלנו אכן הועלו והתעדכנו בשרת.

אם מישהו עשה שינויים בשרת, נרצה "למשוך" אותם אלינו בעזרת הפקודה `git pull` .

לא כל הקבצים צריכים לעלות לשרת

כשאנחנו כותבים קוד, יש המון קבצים שנוצרים. לדוגמה, כשאנחנו עובדים על פרויקט בשפת C++ ומשתמשים ב-Visual Studio - נוצרים לנו בתהליך הקמפול קבצי .exe, .obj. וכו'. אלו קבצים שקשורים להרצה של הפרויקט ולא **לקוד עצמו**. אם אני אשלח לחבר במייל את הקוד שלי – אני לא אשלח את הקבצים האלה, בגלל שהם מיותרים.

בדיוק לשם כך יש ב-git קובץ מיוחד שאנחנו יכולים ליצור שבו נגדיר ל-git אילו קבצים לא נרצה שהוא "יעקוב" אחריהם - קבצים שלא נרצה לשמור / להעביר לאחרים. הקובץ הזה נקרא:

.gitignore

שימו לב! זה לא קובץ רגיל. השם של הקובץ מתחיל בנקודה. אין לקובץ הזה שם, רק סיומת. וודאו שזה בדיוק השם של הקובץ! כל שם אחר לא יעבוד!

המבנה של הקובץ דיי פשוט. נוכל לכתוב באופן ידני את הקובץ הזה (חפשו מדריך באינטרנט), או שנוכל למצוא באינטרנט אחד שמתאים לתוכנת העריכה שבה אנחנו עובדים, ולשפת התכנות שלנו. חפשו gitignore generator בגוגל (או פשוט כנסו ל-www.gitignore.io) ובחרו את שפות התכנות ועורכי הקוד שבהם נשתמש בפרויקט שלנו.

סיכום

עשינו חזרה על כל מה שלמדנו עכשיו, ולמדנו גם כמה דברים חדשים (או לפחות עשינו קצת סדר עם מה שלא היה כל כך ברור עד עכשיו). וודאו שהכול מובן לכם. אם יש משהו לא מובן – קראו אותו שוב. יש המון מקומות שבהם אפשר להיכשל עם גיט, חשוב שתעברו על ההוראות בצורה מדוייקת!

למדנו:

- מה זה מערכת ניהול גרסאות? ומה זה גיט?
- השלבים השונים בשמירה של קובץ
  - Untracked
  - Staged
  - Committed
- להסתכל על הלוג של ה-commit-ים, וגם לחזור אחורה אם צריך.
- להעלות את הקוד לשרת גיט מרוחק.
- להתעלם מקבצים מיותרים בעזרת קובץ gitignore.