

לימוד עצמי 2 – עבודה משותפת

כשעובדים על פרויקט עם אנשים נוספים – יש המון משימות שצריך לעשות. צריך לפתח פיצ'רים חדשים, צריך לתקן באגים קיימים, צריך שמישהו יעבור על הקוד שאחרים כתבו כדי לוודא שהקוד עובד טוב. בקיצור – צריך שתהיה אפשרות לעבוד בו-זמנית על משימות, מבלי להפריע אחד לשני.

אבל עבודה משותפת על קוד יכולה להיות מסובכת. צריך שתהיה לנו דרך לסנכרן את השינויים בין מחשבים שונים, צריך דרך שמאפשרת לעבוד במקביל על הקוד (ולא להיות תלוי אחד בשני), וצריך שתהיה אפשרות לחלק את המשימה הגדולה של "פיתוח פרויקט" למשימות קטנות שאפשר לחלק בין כמה אנשים.

אחת המטרות של מערכת ניהול גרסאות, וג'יט בפרט – היא היכולת לעבוד עם אנשים נוספים על פרויקט אחד. ג'יט יודעת לעשות את זה בצורה מאוד טובה. כמו שאמרנו, הפרויקט הוקם ע"י מייסד לינוקס במטרה ליצור דרך נוחה לניהול בצורה איכותית את הפיתוח של פרויקט Linux – פרויקט שעובדים עליו אלפי אנשים בו זמנית.

סיפור שהיה כך היה

אתמול חברה שלי באה אליי והציעה לי רעיון ממש מגניב לפרויקט – אפליקציה לפלאפון שאפשר לשלוח בה הודעות לחברים. אמרתי לה: "וואי זה בטוח יתפוס! את גאונה! בואי נפתח זריז פרויקט בג'יט ונתחיל לפתח את הפרויקט הזה!". חשבנו על כל מיני פיצ'רים שאנחנו רוצים שיהיו באפליקציה שלנו. רצינו שתהיה אפשרות להוסיף אנשי קשר, שתהיה אפשרות לפתוח צ'אט פרטי עם חבר אחד, ואפשרות לפתוח קבוצה שיהיו בה כמה חברים.

מתחילים לפתח

אני התחלתי לפתח את הפיצ'ר של הקבוצות, וחברה שלי בינתיים התחילה לפתח את הפיצ'ר של פתיחת צ'אט פרטי. כל אחד עבד על הקוד שלו, ואחרי שכל אחד סיים את המשימה, מיזגנו את הקוד של שנינו לפרויקט. אבל הפרויקט עוד לא מוכן להוצאת גרסה לחנות האפליקציות! עוד אין לנו אפשרות הוספת אנשי קשר! חיש מהר רצתי אל המחשב שלי, ופיתחתי במהירות של טייס את הפיצ'ר הזה, ומיזגתי אותו זריז אל הפרויקט.

הבאג הראשון שלי

התחלנו בבדיקות. בדקנו ובדקנו כל מיני מצבי קצה, ופתאום גילינו באג בקוד שלנו. חברה שלי רצה למחשב שלה (הוא היה אצלה בבית, והיא גרה רחוק. מזל שהיא מצטיינת הכיתה בשיעורי ספורט!) ותיקנה את הבאג. גם אצנית, וגם מתכנתת טובה. איזו חברה מוכשרת!

מוציאים גרסה

היא העלתה את השינויים לפרויקט שלנו, ובדקנו שוב שהכול עובד. אחרי המון שעות והמון כוסות קפה, סוף סוף הגענו לגרסה עובדת! טירוף! ישר התקשרנו לגוגל וביקשנו שייעלו את האפליקציה שלנו לחנות האפליקציות.

מתחילים לפתח את הגרסה הבאה

תוך כדי שאני איתם על הקו (סוגר את הפרטים האחרונים) – חברה שלי חשבה כבר על פיצ'ר חדש – שליחת תמונות בהודעה. סיימתי את השיחה עם גוגל והם העלו לנו את האפליקציה לחנות. בזמן שאנחנו נהנים מההצלחה המסחררת של האפליקציה – התחלנו כבר לפתח את הגרסה הבאה. לכו תדעו, אולי בסוף Facebook עוד יקנו אותנו (אמן)...

בכל פרויקט יש הסתעפות

כמו שראינו בסיפור, במהלך עבודה על פרויקט יש המון דברים שאפשר לעשות בו זמנית, ויש שלבים שבהם הכול מתמזג חזרה. יש שלבים בפרויקט שבהם אנחנו מפתחים פיצ'ר, יש שלבים שבהם אנחנו מתקנים באגים, ויש שלבים שבהם אנחנו מוציאים גרסה מוכנה שאפשר לשלוח ללקוח (או למדריך במגשימים, נכון לעכשיו).

בדיוק בשביל זה – יש בגיט קונספט מיוחד שנקרא **branch-ים**. אפשר לעבוד בו זמנית על קוד, מבלי להפריע אחד לשני בגלל שאנחנו פשוט "משכפלים" את כל הקוד הצידה, וממשיכים לפתח אותו במקביל.

המאסטר הקדוש

נתחיל מה-branch המרכזי – שנהוג לקרוא לו **master**, ובו נמצאת הגרסה **המוכנה** של הפרויקט. זו הגרסה ששולחים ללקוח. לכן – נקפיד על כלל מאוד, מאוד, **מאוד חשוב**:

ה-master הוא קדוש!

לא מעלים אליו קוד שלא נבדק, **לא** מעלים אליו קוד שנמצא בשלבי עבודה, **ולא** מעלים אליו קוד שאנחנו לא בטוחים ב-100% שהוא עובד טוב. **המאסטר קדוש!**

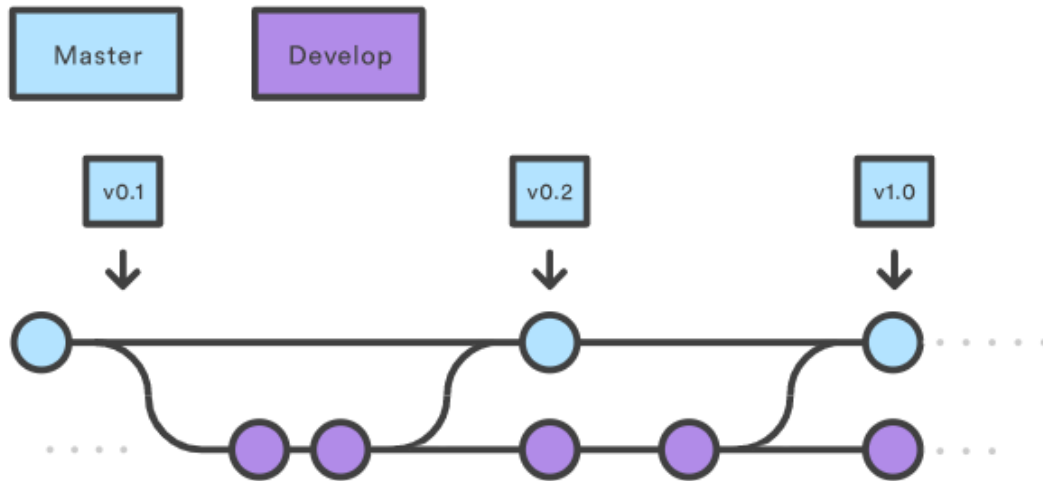
גרסת עבודה - develop

מתוך ה-master, ניצור branch חדש – שבו תהיה הגרסה שעליה אנחנו עובדים. זו הגרסה "המלוכלכת" – אליה כל העבודה שלנו מתמזגת במהלך הפיתוח. גם פיצ'רים חדשים, וגם תיקוני באגים. נהוג לקרוא ל-branch הזה – **develop**, משום שבו מתבצע הפיתוח.

אחרי שנמזג אל תוכו את השינויים שעשינו בפרויקט, נריץ עליו את הבדיקות השונות, ונתקן באגים בהתאם. כשנגיע למצב שבו הקוד שלנו הגיע לרמת גימור גבוהה, והוא עובר את כל הבדיקות שלנו, נוכל לפתוח "בקשת מיזוג" ל-master.

למה פותחים בקשה, ולא ממזגים ישר? כדי שמנהל הפרויקט (המדריך, במקרה שלנו) יוכל לעבור על הקוד ששינינו במהלך העבודה, ולוודא שהקוד באמת איכותי וטוב. אחרי המעבר על הקוד, הוא יוכל למזג את הקוד אל תוך ה-master, ויעלה את הגרסה החדשה לחנות האפליקציות.

כך ייראה הגרף של הפרויקט שלנו (עיגול = commit; צבע = branch):



Branch-ים נוספים

לא תמיד נרצה לעבוד ב-develop! כמו שאמרנו למעלה - יש הסתעפויות שונות בעבודה על פרויקט. לכן מתוך "גרסת העבודה" שלנו (ה-develop) נרצה ליצור הסתעפויות נוספות.

יש כמה מצבים שבהם נהוג לפתוח branch חדש (זו הקונבנציה שאיתה נעבוד במגשימים):

1 – פיתוח פיצ'ר חדש - Feature

פיצ'ר חדש יכול להיות – הוספת אפשרות ליצירת קבוצות צ'אט באפליקציית ההודעות שלי, ופיצ'ר חדש יכול להיות גם סידור של הקוד הקיים (נקרא בשפה המקצועית Refactor). פיצ'ר יכול להיות גדול, ובתוכו יכולים להיות המון פיצ'רים קטנים.

לדוגמה בפיתוח הפיצ'ר "הוספת צ'אט קבוצתי" יש בעצם כמה משימות קטנות יותר:

- עיצוב הממשק שיציג את הקבוצה
- מימוש יצירת קבוצה
- מימוש שליחת הודעה לכל האנשים בקבוצה

לכל תת-משימה נפתח branch נפרד (מתוך ה-branch המרכזי של הפיצ'ר), ובסוף הפיתוח של כל תת-פיצ'ר נמזג אותו חזרה ל-branch של הפיצ'ר המרכזי. בדוגמה שלנו, השמות של ה-branch-ים יכולים להיות:

Feature/add-group-chat

- ➔ Feature/design-gui
- ➔ Feature/create-group
- ➔ Feature/send-message

מתוך ה-branch הראשי (Feature/add-group-chat) יוצאים תת-branch-ים.

שימו לב לשימוש בתחילית "Feature/". נהוג להוסיף אותה לשם של ה-branch כדי לדעת שמדובר ב-branch של פיצ'ר. מוסיפים את ה-slash (/) כדי לסווג את ה-branch תחת ה-branch-ים של פיצ'רים חדשים.

2 – תיקון באג - Bugfix

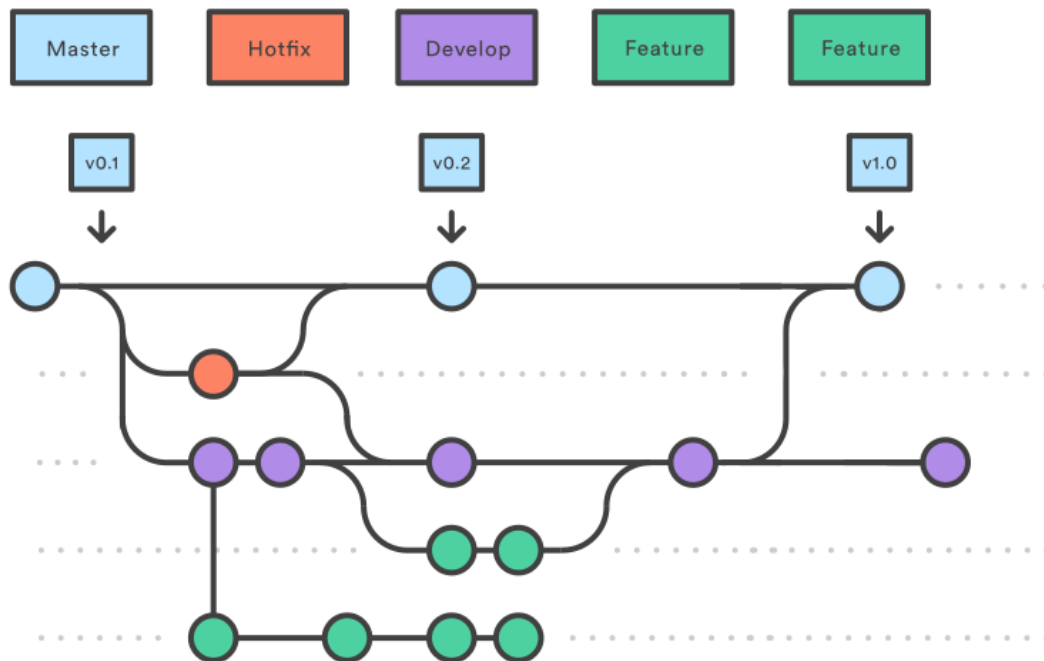
תיקון באג זה לא פיתוח חדש, אלא תיקון של הקוד הקיים. לכן נרצה לפתוח branch עם התחילית "Bugfix/" שבו נתקן את הבאג.

לדוגמה, מצאנו באג - כששולחים הודעה בקבוצה, ההודעה מגיעה לצ'אט פרטי במקום לכל הקבוצה. כדי לתקן אותו – נפתח branch בשם: **Bugfix/group-chat-messages**. אחרי שנתקן את הבאג, נמזג את ה-branch חזרה ל-branch שממנו הוא יצא.

לא פותחים branch בשם **Bugfixes**! זו לא צורה נכונה לעבוד. נפתח branch לכל באג שיש לנו, לא אחד כללי ל-"תיקוני באגים".

3 – תיקון מהיר - Hotfix

נניח והעלנו גרסה ל-master אחרי בדיקות, ופתאום גילינו ששכחנו לעדכן את מספר הגרסה של האפליקציה שמודפסת במסך הראשי. לשם כך, לא צריך לפתוח branch שלם של פיתוח פיצ'ר חדש, אלא יוצרים branch "תיקון מהיר" מתוך ה-master, נעדכן בו את מספר הגרסה, ונפתח שוב בקשת מיזוג ל-master. נשתמש בתחילית **Hotfix/** בתחילת שמו.



סדנה בזוגות - עבודה במקביל

רקע

שלום לכם! אני מייק ואני עובד בחברת Microsoft כמפתח של מערכת ההפעלה האהובה Windows 10. יש לי משימה בשבילכם. כבר הרבה שנים שלא שינינו שום דבר ב-cmd המפורסם. הפקודות נשארו בדיוק אותו הדבר מאז Windows 95 (שהיה מאוד מוצלח בזמנו, דרך אגב). היום המציאות השתנתה, והחלטנו להוסיף ל-cmd כמה פקודות חדשות שיעזרו לאנשים שמשתמשים ב-cmd לבצע כל מיני משימות יותר בקלות.

אז חילקנו את הפרויקט בין העובדים אצלנו וכל אחד קיבל פקודה חדשה שהוא צריך לכתוב, והכול הלך ממש סבבה. עד שלפני חודש – שני העובדים שהיו אמורים לכתוב את אחת הפקודות התפטרו (בלינוקס זה לא היה קורה...) ונשארו בלי מפתחים. שמענו שאתם תותחים ב-C++ (וכמובן כוח עבודה זול) אז נשמח אם תוכלו לעזור לנו עם הקוד הזה. תודה רבה חניכי מגשימים יקרים!

מה אנחנו עומדים לפתח?

שאלנו את המשתמשים שלנו: "לאילו דברים אתם משתמשים במחשב שלכם?" ודבר אחד שחזר אצל הרבה היה – "עבודה על פרויקטים". לכל אדם יש פרויקטים שעליהם הוא עובד. מה גם שאנשים שנמצאים בתפקיד ניהולי – שומרים בנוסף גם את הפרויקטים שהעובדים שלהם עושים.

אז ישבנו פה כל החברה ב-Microsoft וחשבנו על פקודה חדשה שתעזור למשתמש לעקוב אחר הפרויקטים שלו. בעצם אנחנו הולכים לממש פקודת cmd של "מנהל פרויקטים"!

המשתמש יוכל להוסיף לשם את הפרויקטים שעליהם הוא עובד, למחוק פרויקט שהוא סיים, ובגדול – לנהל רשימה של כל הפרויקטים שלו. איזה מגניב, הא? אשכרה פקודה שהולכת להתווסף ל-cmd הרשמי של Microsoft!

ככה התפריט של הפקודה הולך להיראות:

```
Projects Manager - v0.0.1
-----

new <project-name> <author>
show <?project-name>
delete <project-name>
modify <project-name> <new-name>
```

הוראות מקדימות

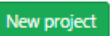
- בסדנה הזו אנחנו **הולכים לעבוד בזוגות**. חלק מהמשימות נעשה **במקביל** על 2 מחשבים, וחלק נעשה ביחד על מחשב אחד. מצאו בעזרתו של המדריך שלכם חסר נוסף שסיים את משימת הלימוד הראשונה, ועבדו ביחד.

- **עיקר הדגש בפרויקט הוא על עבודה נכונה עם git** – הקוד שלכם הולך להיכנס לפרויקט ענק של חברת Microsoft – אז הוא חייב להיות מעולה! לכן, **אסור להשתמש בממשק גרפי של גיט במהלך התרגיל!** המטרה היא שתלמדו לעבוד עם הפקודות עצמן, כבסיס להבנה של אופן הפעולה של גיט. אל תדאגו, בהמשך הסמסטר נלמד לעבוד עם כלי גרפי.
- בסוף הסדנה, תצטרכו להגיש את העבודה שלכם למדריך. אם לא הספקתם לסיים אותה בכיתה, תצטרכו להמשיך אותה בבית (ביחד עם בן הזוג). בסוף התרגיל יוסבר בדיוק מה צריך להגיש.
- אם אתם לא בטוחים איזו פקודה של גיט צריך להריץ בכל שלב, או אילו דגלים יש לה – תשתמשו בדגל **-h** או בדגל **--help** כדי ללמוד עוד על הפקודה. בנוסף, תוכלו לפתוח Repo נוסף על המחשב לתרגול במידה ואתם לא בטוחים מה הפקודה שאתם עומדים להריץ הולכת לעשות.
- אם הגעתם למצב שבו אתם מרגישים שממש "דפקתם" את ה-Repo שלכם – יש כמה דברים שאתם יכולים לעשות:
 - להריץ `git status` ולנסות להבין מה קרה
 - להריץ `git log` ולנסות להבין מה קרה, ואיך חוזרים אחורה
 - להסתכל ב-GitLab על ה-commit-ים האחרונים, או על ה-branch-ים שפתחנו
 - הכנו בשבילכם מסמך שמסביר איך לפתור בעיות שאתם עלולים להיתקל בהם – [סדנת גיט – פתרון בעיות](#).

יאללה מוכנים? #התחלנו

שלב 1 – פותחים פרויקט

פתיחת Repository

נתחיל מפתיחת **Git Repository** (מעכשיו נקרא לו – repo) חדש על **שרת git**. במקרה שלנו נשתמש בשירות החינמי GitLab.com. התחברו לאתר עם המשתמש שלכם. לחצו על כפתור  , וכנסו לדף הוספת פרויקט.

Project name: learn-git

Project description: Learning git in Magshimim.

Visibility Level: Private

Initialize repository with a README: Yes

נלחץ על  ונועבר לדף של הפרויקט.

מדהים! יש לנו repo חדש בשרת של GitLab.com, ויש בתוכו כבר את הקובץ README.md.

זהו קובץ לא חובה, אך הוא שימושי. הוא מוצג כשנכנסים לדף של ה-repo באתר (כרגע הוא מכיל את שם הפרויקט ואת התיאור שלו). הפורמט שבו הקובץ בנוי נקרא Markdown, והוא דיי פשוט. ניגע בזה קצת בעתיד.

הוספת Member

כנסו לדף Members > Settings באתר, והוסיפו את בן/בת הזוג שלכם כחבר צוות. הוסיפו גם את היוזר של קורס עקרונות (כדי שהמדריך יוכל לבדוק את ההגשה שלכם) – **Ekronot** (אפשר גם באמצעות המייל mag.ekronot@gmail.com).

GitLab member or Email address: <partner-username>

Choose a role permission: Maintainer

Access expiration date: -null-

הוספת קובץ gitignore

נחפש **Gitignore Generator** בגוגל (או ניכנס ל-www.gitignore.io) ונבחר את שפות התכנות ועורכי הקוד שבהם נשתמש בפרויקט שלנו – **Visual Studio** ו**שפת ++C**.

נוסיף את הקובץ שנוצר ע"י לחיצה על כפתור ה- **[+]** בפרויקט שלנו באתר, ונוסיף קובץ חדש בשם "**gitignore**". עם התוכן שייצרנו.

תודה: שיש נקודה בתחילת השם! זה חשוב מאוד! אחרת – גיט לא יכיר אותו.

תודה: שהקובץ שלכם תקין, ושהוא באמת מתאים ל-Visual Studio וגם לשפת ++C.

עושים Clone ל-Repo

אחרי שיצרנו repo על השרת (נקרא לו מעכשיו Remote Repo), נרצה "לשכֵּט" אותו למחשבים של שנינו. לשם כך נעשה נרצה לעשות לו **git clone** לתוך תיקייה מקומית על המחשב.

עשו **כל אחד בנפרד** Clone לפרויקט שבאתר אל תוך תיקייה על המחשב. הפקודה אמורה להיראות **בערך** ככה (כמובן צריך לשנות את הקישור לקישור של הפרויקט שלכם):

```
git clone https://gitlab.com/magshigit/learn-git.git
```

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

נפתח cmd חדש בתוך תיקיית הפרויקט, ונריץ בו **git status**, אמור להיות מודפס:

```
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

לפני שאנחנו מתחילים לעבוד על הפיתוח של הפרויקט, יש עוד כמה דברים שצריך לעשות:

פותרים branch עבודה

כמו שדיברנו למעלה, ה-**master** הוא קדוש! לכן כדי להתחיל לפתח את הפרויקט, נרצה ליצור branch חדש בשם **develop**, לפתוח בו פרויקט ב-Visual Studio, ולהעלות לשרת.

נפתח branch חדש בעזרת הפקודה `git branch <branch-name>`.

פתחו branch בשם **develop**.

נריץ `git branch` כדי לראות אילו branch-ים יש לנו בפרויקט כרגע:

```
develop
* master
```

ב-**ירוק** (עם * בהתחלה) מסומן ה-branch שבו אנחנו נמצאים כרגע (master). נרצה לעבור ל-develop. לשם כך, נריץ את הפקודה `git checkout <branch-name>` (ובשם של ה-branch נרשום develop). הפקודה הזו "עוברת" בין branch ל-branch (אפשר גם להשתמש בה כדי לעבור מ-commit ל-commit כמו שראינו במסמך לימוד עצמי 1).

נריץ שוב `git branch`:

```
* develop
master
```

מעולה! עכשיו כל שינוי שנעשה בתיקייה שלנו ישתנה ב-develop, אבל לא ב-master. גם commit-ים חדשים שניצור יישמרו תחת develop ולא תחת master.

ווידוא תקינות של ה-gitignore

חשוב לוודא שקובץ ה-**gitignore** שלנו עובד טוב ומתעלם מהקבצים. צרו קובץ בשם test.exe וגם תיקייה בשם Debug בתיקיית הפרויקט שלכם. הריצו `git status` ותוודאו שהוא נקי! אם הקובץ test.exe מופיע תחת Untracked Files – **סימן שעשיתם משהו לא נכון**. חזרו על ההוראות של יצירת קובץ "**gitignore**". פעם נוספת.

אם הכול תקין – **כל הכבוד!** מחקו את הקובץ והתיקייה ותמשיכו הלאה.

העלאת ה-branch לשרת

ב-local repo שלנו אנחנו יכולים ליצור איזו הסתעפות שבא לנו בפרויקט. אפשר לפתוח branch כדי לנסות משהו קטן ואז ישר למחוק אותו, או שאפשר לפתוח מלא branch-ים סתם בשביל הכיף. אבל לא נרצה שכל שינוי מקומי או כל branch שאנחנו פותחים על המחשב – יהיה גם על השרת! נרצה להעלות רק branch-ים שאנחנו רוצים שיהיו לכולם.

לכן בגיט יש שני סוגי branch-ים – **מקומיים** ו-**מרוחקים** (Local vs. Remote).

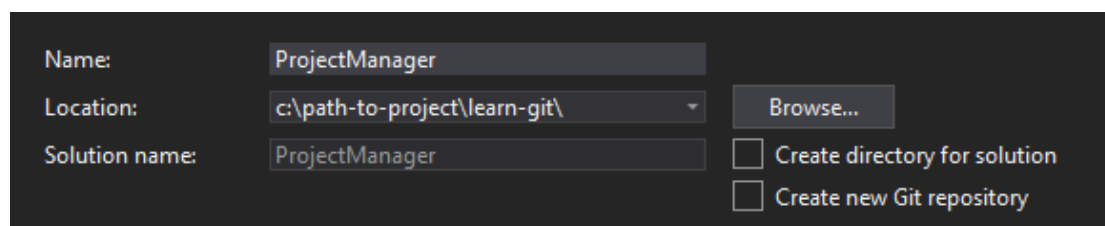
כשאנחנו פותחים branch חדש על המחשב ועובדים עליו, צריך לדחוף אותו גם לשרת. בשביל לעשות את זה, צריך שיהיו בו שינויים כלשהם (commit אחד או יותר) כדי שלגיט יהיו שינויים לדחוף.

בשביל זה, ניצור שינוי כלשהו. אפשר סתם ליצור קובץ ולעשות לו commit, אבל עדיף שננצל את ההזדמנות לשינוי מועיל:

פתיחת פרויקט ב-Visual Studio

פתחו פרויקט חדש ב-Visual Studio בתוך התיקייה של ה-repo. שם הפרויקט יהיה **"ProjectManager"**. וודאו שה-checkbox שנמצא במסך יצירת הפרויקט "Create new Git"

repository "לא מסומן" (כי כבר יש לנו repo פתוח). הורידו גם הסימון בתיבה Create directory for solution", כי כבר יש לנו תיקייה לפרויקט, אין צורך בעוד אחת.



הוסיפו בתוך הפרויקט קובץ **Main.cpp**, והדפיסו בו "שלום עולמי" (Hello My World) וגם את הגרסה הנוכחית של הפרויקט – v0.0.1.

חשוב מאוד: וודאו שיש בתיקייה הראשית של ה-Repo **קובץ .gitignore**. שמתעלם מקבצים מיותרים של Visual Studio. אנחנו מדגישים את זה בכוונה, כי טעות עם הקובץ תהיה קשה לתיקון בהמשך.

אם תריצו `git status` הפלט חייב להיות כזה:

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

    ProjectManager/

nothing added to commit but untracked files present (use "git add" to track)
```

אחרי שנריץ `git add ProjectManager/` (אפשר לעשות add גם לתיקייה שלמה)

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

    new file:   ProjectManager/Main.cpp
    new file:   ProjectManager/ProjectManager.sln
    new file:   ProjectManager/ProjectManager.vcxproj
    new file:   ProjectManager/ProjectManager.vcxproj.filters
```

נשאר לנו רק לעשות commit לשינויים (לא לשכוח הודעה אינדיקטיבית!) ולעשות להם push לשרת. כשנגסה לעשות `git push` תודפס לנו שגיאה:

```
fatal: The current branch develop has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin develop
```

השגיאה אומרת שאין branch בשם develop על השרת! נו ברור... הרי לא יצרנו אותו על השרת, יצרנו אותו רק על ה-Local Repo!

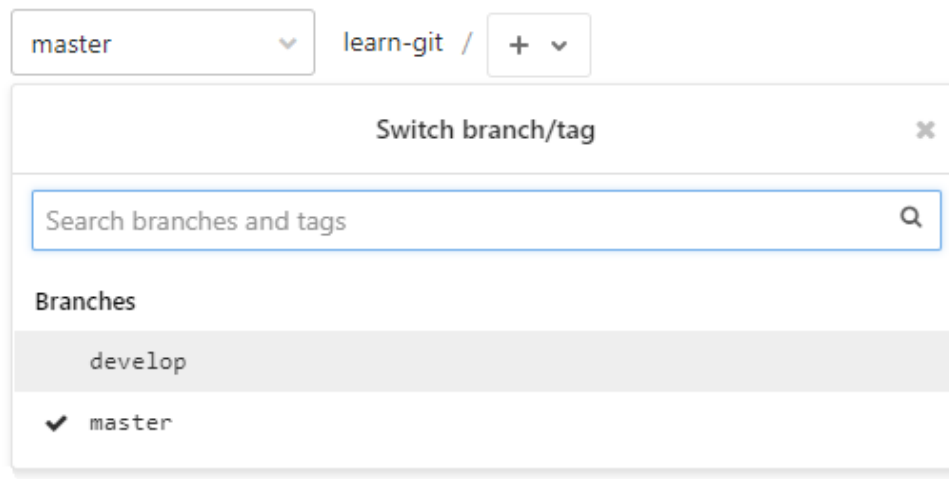
נריץ את הפקודה: `git push --set-upstream origin develop`.

הדגל `--set-upstream` מייצר branch מרוחק בשם של ה-branch המקומי. (יווצר branch בשם develop על השרת).

המילה origin משמעותה "המקור", "ההתחלה". הפרמטר הזה מייצג את ה-Remote Repo שאליו אנחנו רוצים לדחוף את השינויים. אנחנו תמיד נשתמש במילה origin כדי להגיד "דחוף את השינויים ל-Repo שעשיתי לו Clone".

אם תמיד נכתוב origin בפרמטר השני, אז למה בכלל יש אותו? כי תיאורטית - אפשר לדחוף שינויים שעשינו ל-Remote Repo אחר. אבל אנחנו לא הולכים להשתמש באפשרות הזאת. את השינויים שלנו נרצה תמיד לדחוף ל-Remote Repo שלנו, ולא לאחר. גם מחוץ למגשימים לא משתמשים בפיצ'ר הזה כמעט אף פעם. לכן תמיד נכתוב **origin**.

נוודא באתר שנוצר branch חדש, ושהקוד שלנו נמצא בתוכו:



במחשב השני – הריצו `git pull` כדי למשוך את ה-branch החדש,

ועברו אליו בעזרת `git checkout develop`.

יופי! עכשיו סוף סוף אפשר להתחיל לעבוד על הקוד שלנו. אמנם לקח לנו הרבה זמן לפתוח פרויקט חדש, אבל זה רק בגלל שאנחנו מתחילים ב-git ולמדנו עקרונות בסיסיים על הדרך. בפעמים הבאות שתפתחו פרויקט זה יהיה הרבה יותר מהיר... אז אל דאגה!)

שלב 2 – מפתחים את הקוד הראשי

בשלב הזה נתחלק ל-2 מחשבים. המשימות בשלב זה יתחלקו למשימות **שעושים במקביל**, ומשימות **שעושים ביחד**. כל משימה תסומן בהתאם. תחלקו ביניכם את המשימות במקביל, ותעבדו עליהן בו זמנית משני מחשבים.

שימו לב: צריך לסיים את כל המשימות המקבילות לפני שמתחילים משימה שעושים ביחד. המלצה: תקראו גם את המשימות של בן/ת הזוג שלכם. ככה תוכלו להבין יותר את הפרויקט שלכם. בנוסף, כל אחד בפרויקט אמור להכיר את הקוד של האחרים.

- **עוד דבר חשוב:** בתרגילים הקרובים נעבוד מתוך develop –

משימה במקביל - הוספת הדפסת תפריט

שנו את ה-main שלכם שיקרא לפונקציה `printMenu` – שמדפיסה את התפריט הבא (זו תיבת טקסט, לא תמונה. מוזמנים להעתיק ממנה):

```
Projects Manager - v0.0.1
-----

new <project-name> <author>
show <?project-name>
delete <project-name>
modify <project-name> <new-name>
```

שמרו את השינויים (add & commit) ועשו להם push.

וודאו שאתם עושים push ל-develop ולא ל-master!

משימה במקביל - הוספת המחלקה Project

צרו מחלקה חדשה בשם Project וממשו אותה.

Project
- m_name : std::string - m_author : std::string
+ Project (std::string name, std::string author) + std::string getName () + std::string getAuthor () + std::string toString ()

שמרו את השינויים (add & commit) ועשו להם push.

וודאו שאתם עושים push ל-develop ולא ל-master!

משימה במקביל - הוספת המחלקה ICommand

כדי לממש את "פקודות cmd" במערכת, ניצור מחלקת interface שמייצגת Command – פקודה בודדת.

<<Interface>> ICommand
+ doAction(std::vector<Project>& projects) = 0 : void

כל מחלקה יורשת חייבת לממש את הפעולה doAction על האובייקט projects. זה יתבהר יותר בהמשך. שמרו את השינויים (add & commit) ועשו להם push.

משימה במקביל - הוספת הפונקציה commandInvoker

הפונקציה הזו תיקרא מה-main. היא אחראית על קריאה לפקודה (Command) המתאימה, בהתאם לפקודה שהמשתמש הכניס כקלט.

```
void commandInvoker(const std::string& command,
                    std::vector<Project>& projects)
```

הפונקציה תפריד את command ל- `std::vector<std::string>` ע"פ התו רווח (" "). ותממש switch (יותר נכון - if-else, כי switch לא עובד על `std::string`) שיידע לקרוא לפונקציה שתטפל בכל סוג פקודה (new, modify, delete וכו'). כרגע הפונקציה לא תקרא לאף פקודה, נסיף את זה בהמשך.

שמרו את השינויים (add & commit) ועשו להם push.

וודאו שאתם עושים push ל-develop ולא ל-master!

משימה במקביל - הוספת לולאה ל-main

הוסיפו לולאה ל-main שמקבלת קלט של `std::string` מהמשתמש (הפקודה שהוא רוצה להריץ), כל עוד לא הוקלדה הפקודה "exit".

עטפו את הקוד ב-try-catch כדי לתפוס שגיאות של קלט מהמשתמש.

שמרו את השינויים (add & commit) ועשו להם push.

וודאו שאתם עושים push ל-develop ולא ל-master!

משימה ביחד - מיזוג הקוד

עבדנו ד"י הרבה בנפרד, ועכשיו הגיע הזמן למזג את העבודה, כדי שאצל שנינו תהיה הגרסה המלאה של הפרויקט, וכדי שנוכל להתפצל מכאן ל-branch-ים נפרדים להמשך עבודה.

וודאו שאין לכם שינויים מקומיים (`git status` אמור להיות נקי), והריצו `git pull` משני המחשבים.

תוודאו שאצל שניכם יש **בדיוק** את אותו הקוד בתיקייה. אם אתם לא מצליחים למזג ומקבלים משהו שנראה כמו השגיאה הבאה:

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From gitlab.com:Magshimim/my-project
* branch develop -> FETCH_HEAD
a608e7b..41af65d master -> origin/develop
Auto-merging Main.cpp
CONFLICT (content): Merge conflict in Main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

סימן שיש לכם קונפליקט בקוד. כנראה שניכם ערכתם את אותם שורות קוד, ומנגנון המיזוג של גיט לא יודע איזה שינוי אתם רוצים להשאיר (או אולי אתם רוצים להשאיר את שניהם?). כדי לפתור את הקונפליקט תוכלו להשתמש במדריך שהכנו לכם - [דנת Git – פתרון בעיות](#). כעת בשני המחשבים אמור להיות בדיוק אותו הקוד. איזה כיף!

משימה ביחד - הוספת אובייקט `ICommand*` ל-`commandInvoker`

במשימה הבאה נתחיל לממש את המחלקות שירשו מ-`ICommand` ומממשות את הלוגיקה של "ביצוע הפקודה". אבל לפני כן, ממשו ביחד בראש הפונקציה `commandInvoker` את ההכרזה על משתנה מסוג `ICommand*`, ושימו בה `nullptr`.

אחרי בלוק ה-`if-else`, (שעוד מעט נוסיף לו מימוש של יצירת אובייקט חדש מטיפוס שירש את `ICommand`, ושם אותו בתוך המשתנה שיצרנו) הוסיפו קריאה לפונקציה `doAction/projects` על האובייקט שיצרנו.

המבנה של הפונקציה אמור להיות בערך כזה:

1. Split "command" by the char Space
2. Declare variable of type `ICommand*` named `commandObject`
3. If (`command == "new"`)
→ Create new `ICommand` object, and assign to `commandObject`
4. If (`command == ...`)
→ ...
5. Call `doAction` on `commandObject`

הוסיפו ביחד את המימוש הזה לפונקציה `commandInvoker`, שמרו את השינויים (`& add` `commit`) ועשו להם `push`. אל תשכחו לעשות `pull` מהמחשב השני.

וודאו שאתם עושים `push` ל-`develop` ולא ל-`master`!

משימה ביחד - חלוקת משימות, והתפצלות

אחרי שפיתחנו את הבסיס של המערכת, הגיע הזמן לפתח את הלוגיקה שמטפלת בפקודות עצמן. נרצה לממש 4 מחלקות:

- `CommandNew` – מטפלת ביצירה של פרויקט חדש, ומוסיפה אותו למשתנה `projects` מסוג `vector<Project> :std`.
 - `CommandShow` – מטפלת בהדפסה למסך של פרויקט מתוך המשתנה `projects`, או כל הפרויקטים – אם לא מקבלת שום פרמטר.
 - `CommandDelete` – מוחקת פרויקט מתוך המשתנה `projects`.
 - `CommandModify` – מעדכנת פרויקט מתוך המשתנה `projects`.
- למימוש של כל מחלקה, נפתח branch חדש מתוך `develop`, ונמזג אותו חזרה אחרי שנסיים.

לפני תחילת העבודה, חלקו ביניכם את המשימות שמופיעות בדף הבא, וצרו branch חדש של Feature מתוך develop לפני תחילת כל משימה (צריך לעבור חזרה ל-develop ולפתוח ממנו branch חדש).

משימה במקביל - מימוש מחלקת CommandNew

פתח branch חדש מתוך develop בשם **Feature/command-new**.

הוסיפו את המימוש של המחלקה:

CommandNew : ICommand
- m_projectName : std::string - m_projectAuthor : std::string
+ CommandNew (const std::string & projectName, const std::string & projectAuthor) + doAction(std::vector<Project>& projects) : void

הוסיפו את היצירה של האובייקט בפונקציה `commandInvoker` (בתוך התנאי הרלוונטי בבלוק ה-if-else), והעבירו לאובייקט את הפרמטרים הרלוונטיים מהקלט שקיבלנו מהמשתמש.

שמרו את השינויים (`add & commit`) ועשו להם `push` **לתוך ה-branch שפתחתם**.

וודאו שאתם לא דוחפים שינויים לתוך develop!

משימה במקביל - מימוש מחלקת CommandShow

פתח branch חדש מתוך develop בשם **Feature/command-show**.

הוסיפו את המימוש של המחלקה:

CommandShow : ICommand
- m_projectName : std::string
+ CommandShow () + CommandShow (const std::string & projectName) + doAction(std::vector<Project>& projects) : void

אם הפקודה לא מקבלת שם של פרויקט – היא מדפיסה את כל הפרויקטים.

הוסיפו את היצירה של האובייקט בפונקציה `commandInvoker` (בתוך התנאי הרלוונטי בבלוק ה-if-else), והעבירו לאובייקט את הפרמטרים הרלוונטיים מהקלט שקיבלנו מהמשתמש.

שמרו את השינויים (`add & commit`) ועשו להם `push` **לתוך ה-branch שפתחתם**.

וודאו שאתם לא דוחפים שינויים לתוך develop!

משימה במקביל - מימוש מחלקת CommandDelete

פתח branch חדש מתוך develop בשם **Feature/command-delete**.

הוסיפו את המימוש של המחלקה:

CommandDelete : ICommand
- m_projectName : std::string
+ CommandDelete (const std::string & projectName)
+ doAction(std::vector<Project>& projects) : void

הוסיפו את היצירה של האובייקט בפונקציה `commandInvoker` (בתוך התנאי הרלוונטי בבלוק ה-if-else), והעבירו לאובייקט את הפרמטרים הרלוונטיים מהקלט שקיבלנו מהמשתמש.

שמרו את השינויים (add & commit) ועשו להם push **לתוך ה-branch שפתחתם**.

וודאו שאתם לא דוחפים שינויים לתוך develop!

משימה במקביל - מימוש מחלקת CommandModify

פתח branch חדש מתוך develop בשם **Feature/command-modify**.

הוסיפו את המימוש של המחלקה:

CommandModify : ICommand
- m_projectName : std::string
- m_newName : std::string
- m_newAuthor : std::string
+ CommandModify (const std::string & projectName, const std::string & newName const std::string & newAuthor)
+ doAction(std::vector<Project>& projects) : void

הוסיפו את היצירה של האובייקט בפונקציה `commandInvoker` (בתוך התנאי הרלוונטי בבלוק ה-if-else), והעבירו לאובייקט את הפרמטרים הרלוונטיים מהקלט שקיבלנו מהמשתמש.

שמרו את השינויים (add & commit) ועשו להם push **לתוך ה-branch שפתחתם**.

וודאו שאתם לא דוחפים שינויים לתוך develop!

משימה ביחד - מיזוג כל ה-branch-ים ל-develop

אחרי שעבדנו על הקוד שלנו, אמורים להיות לנו 4 branch-ים חדשים בפרויקט שלנו (מלבד master ו-develop).

כנסו באתר לִדף **Branches** (שנמצא בתפריט מצד שמאל). זה בערך מה שאתם אמורים לראות:

Active branches				
Y develop f6c7e27a · Commit mes...	0 3	Merge request	Compare	
Y Feature/command-show d9174aeb · Commit mes...	0 1	Merge request	Compare	
Y Feature/command-new ffa590ca · Commit mes...	0 4	Merge request	Compare	
Y Feature/command-modify cb678a9c · Commit mes...	0 1	Merge request	Compare	
Y Feature/command-delete ba36fcd7 · Commit mes...	0 2	Merge request	Compare	

עכשיו הגיע הזמן שנמזג את כל ה-branch-ים שפתחנו אל תוך ה-develop.

כדי לעשות את זה, נלחץ על הכפתור **Merge Request** של אחד ה-branch-ים החדשים שיצרנו, ונועבר לִדף **New Merge Request**. אנחנו הולכים לפתוח "בקשת מיזוג" (נקרא **Merge Request**) מ-branch אחד ל-branch אחר.

בראש הדף רשום ה-Source Branch וה-Destination Branch:

New Merge Request

From **Feature/command-new** into **master** [Change branches](#)

אנחנו לא רוצים לפתוח בקשת מיזוג ל-master, אלא נרצה לפתוח בקשה ל-develop. לכן

נלחץ על הקישור **Change branches**, ונשנה ל-develop. נלחץ על **Compare branches and continue** כדי לאשר את ההחלפה, וזה יחזיר אותנו לִדף הקודם.

נוריד את התחילית **"WIP: "** בתיבה **Title** (המשמעות של זה היא Work In Progress, אבל מאחר שאנחנו סיימנו את העבודה על ה-branch - זה לא רלוונטי).

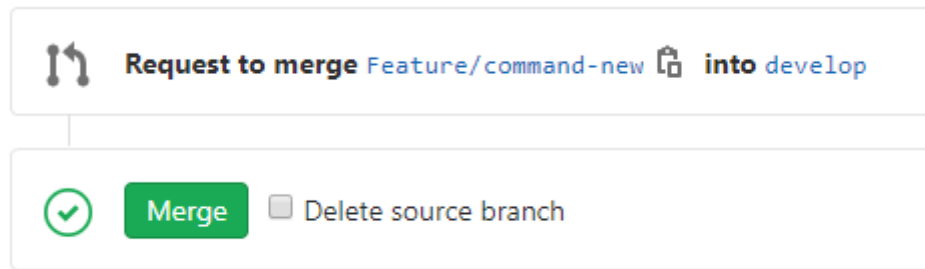
חשוב: נשאיר את השדות בדף עם הערך הדיפולטי שלהם, אבל **נוריד את הסימון** בתיבה: "Delete source branch when merge request is accepted" כדי שה-branch לא יימחק (והמדריך יוכל לראות אותו בהגשה).

Merge options ☐ Delete source branch when merge request is accepted.

נלחץ על **Submit merge request** כדי לאשר את הפתיחה של בקשת מיזוג חדשה.

אם הכול עבד כמתוכנן, אנחנו אמורים להגיע לִדף שבו מופיעה בקשת המיזוג שלנו.

בראש הדף נראה את התיבה הבאה:



נלחץ על הכפתור **Merge** כדי לאשר את "בקשת המיזוג" שלנו, והשינויים ימוזגו לתוך `develop`.

עברו ל-**develop** ב-Local Repo שלכם, ועשו ממנו **pull** בשני המחשבים.

בדקו ב-`log` שהשינויים שמוזגו אכן קיימים אצל שניכם בתוך **develop**.

מזגו גם את שלושת ה-**branch**-ים האחרים באותה הדרך.

שלב 3 - פתיחת בקשת מיזוג ל-`master`

אחרי שכל השינויים שלנו נכנסו ל-`develop`, נוודא שהקוד שלנו רץ ועובד.

הריצו את הקוד על אחד המחשבים, ותקנו באגים קטנים אם יש.

אחרי שהגענו לגרסה שעובדת, הגיע הזמן לפתוח בקשת מיזוג (Merge Request) מ-`develop` ל-`master`.

אימאלה!!!

כל מיזוג ל-`master` הקדוש הוא אירוע גדול בחיים של מפתח. קחו רגע כדי לנשום ולהירגע לפני שאתם ממשיכים.

פתחו Merge Request בדף "**Branches**" באתר מתוך ה-`develop` אל `master`.

שימו לב – אל תלחצו על Merge אחרי שיצרתם את בקשת המיזוג! אתם לעולם לא מאשרים בקשת מיזוג ל-`master`. מי שיכול לעשות את זה הוא רק המדריך שלכם, אחרי שהוא עבר על הקוד שלכם!

מה צריך להגיש למדריך?

שניכם אמורים להגיש **קישור ל-Merge Request שלכם**. הקישור אמור להיראות בערך ככה:

https://gitlab.com/magshigit/learn-git/-/merge_requests/5

זהו! סיימנו! איזה כיף. אתם יכולים לחזור לפרויקט גלריה, ולהמשיך את המשימות שלו.

בהצלחה!