תמונה שמכילה גרפיקה, אומנות קליפיפם, לוגו, סמל

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

הכשרת בדיקות אוטומטיות







# תוכן עניינים

תוכן

[תוכן עניינים 2](#_Toc201672301)

[מבוא 5](#_Toc201672302)

[לפני שנתחיל 6](#_Toc201672303)

[סביבת העבודה – IDE 6](#_Toc201672304)

[Node.JS 6](#_Toc201672305)

[התקנת Git 6](#_Toc201672306)

[ספריות NPM 6](#_Toc201672307)

[רשימת הספריות 7](#_Toc201672308)

[Git 9](#_Toc201672309)

[מבוא 9](#_Toc201672310)

[הכשרה 9](#_Toc201672311)

[GitHub 10](#_Toc201672312)

[מבוא 10](#_Toc201672313)

[הכנת סביבת העבודה 10](#_Toc201672314)

[דחיפת השינויים לשרת המרוחק 11](#_Toc201672315)

[Pull Request 11](#_Toc201672316)

[Git Rebase 12](#_Toc201672317)

[שפת typescript 13](#_Toc201672318)

[מבוא 13](#_Toc201672319)

[הכשרה 13](#_Toc201672320)

[חומר קריאה 13](#_Toc201672321)

[תרגולים 14](#_Toc201672322)

[Postman ובקשות API 15](#_Toc201672323)

[מבוא 15](#_Toc201672324)

[בקשות API 15](#_Toc201672325)

[תוכנת ה Postman 17](#_Toc201672326)

[סיכום קצר 18](#_Toc201672327)

[תרגול 19](#_Toc201672328)

[בקשות 19](#_Toc201672329)

[אלמנטים באתר האינטרנט 20](#_Toc201672330)

[מבוא 20](#_Toc201672331)

[חלונית ה inspect 20](#_Toc201672332)

[מציאת זיהוי ייחודי 21](#_Toc201672333)

[הכשרה 22](#_Toc201672334)

[Design patterns בבדיקות 23](#_Toc201672335)

[תבניות עיצוב בבדיקות 24](#_Toc201672336)

[POM 24](#_Toc201672337)

[POR 24](#_Toc201672338)

[הערת צד אחרונה לפני שנתחיל 24](#_Toc201672339)

[Playwright 25](#_Toc201672340)

[לפני לפני שנתחיל – דף קונבנציות 25](#_Toc201672341)

[קונבנציות להכשרה – קבצי TS 25](#_Toc201672342)

[קראו אותי כשתתחילו לכתוב קוד 26](#_Toc201672343)

[מבוא 27](#_Toc201672344)

[לפני שנתחיל 27](#_Toc201672345)

[חומר למידה 27](#_Toc201672346)

[תרגול 27](#_Toc201672347)

[הכשרה 29](#_Toc201672348)

[המערכת שעליה נכתוב בדיקות 29](#_Toc201672349)

[התשתית 29](#_Toc201672350)

[הסבר קצר על התשתית 29](#_Toc201672351)

[תרגול 33](#_Toc201672352)

[משימה ראשונה 33](#_Toc201672353)

[משימה שנייה 33](#_Toc201672354)

[משימה שלישית 34](#_Toc201672355)

[משימה רביעית 34](#_Toc201672356)

[Cucumber 36](#_Toc201672357)

[לפני לפני שנתחיל – דף קונבנציות 36](#_Toc201672358)

[קונבנציות להכשרה - קבצי feature 36](#_Toc201672359)

[קונבנציות להכשרה – קבצי step-definitions 37](#_Toc201672360)

[כללי ניסוח לצעד 37](#_Toc201672361)

[מבוא 38](#_Toc201672362)

[הכשרה 38](#_Toc201672363)

[התשתית 38](#_Toc201672364)

[הסבר קצר על התשתית 38](#_Toc201672365)

[תרגול 46](#_Toc201672366)

[משימה ראשונה 46](#_Toc201672367)

[משימה שנייה 46](#_Toc201672368)

[משימה שלישית 46](#_Toc201672369)

# מבוא

קודם כל, ברוכים הבאים להכשרה הבלמסית של כתיבת בדיקות אוטומטיות, אנחנו שמחים שאתם כאן איתנו 😊

מסמך זה נועד ללמד אתכם את כל הכלים הנדרשים, על מנת להיות הבודקים הכי טובים בתחום.

כפי שראיתם בכותרת המסמך, אנו נלמד את הכלים הבאים:

1. Typescript
2. playwright
3. Cucumber

Typescript הינה שפת התכנות שאיתה נכתוב את הקוד שלנו.

Playwright הוא כלי הבדיקות שלנו, כל הפעולות שהמשתמש מבצע, כמו לחיצה על כפתור, הזנת טקסט לתוך תיבת טקסט, יתבצע עם הספרייה הזאת.

Cucumber הוא כלי הניהול שלנו. לבדיקות אוטומטיות יש כמה מתודולוגיות, אחת מהן נקראת BDD (Behavioral Driven Development), מתודולוגיה זו באה לקראת המשתמש, היא מאפשרת לו לכתוב קוד באופן שכל בן אדם יוכל לקרוא ולהבין מה הבדיקה עושה.

אנו נרחיב על כל חלק בהמשך המסמך, מוזמנים לקרוא קצת באינטרנט יותר על כל כלי.

כמובן, שנלמד גם על git, חיפוש אלמנטים בדף האינטרנט, בקשות API ועוד

אז מבלי יותר מדי עיכובים, בואו נתחיל לבדוק 😎

# לפני שנתחיל

בשביל שנתחיל לכתוב קוד כמו בודקים מקצועיים, נצטרך להתקין כמה דברים למחשב.

## סביבת העבודה – IDE

אני אישית מעדיפה סביבות עבודה של JetBrains, שניים שמומלצים לשימוש הם:

קישור ל Aqua – IDE שנועד לבודקים, בשביל בודקים:

<https://www.jetbrains.com/aqua/download/#section=windows>

קישור ל WebStorm – IDE ממש נחמד לתכנות ב typescript:

<https://www.jetbrains.com/webstorm/download/#section=windows>

כמובן, שיש עוד המון IDE's שאתם יכולים להתקין, כמו Visual Studio Code לדוגמא, ולהוסיף לו תוספים של cucumber, ו typescript.

## Node.JS

היא סביבת הרצה ל JS ול TS, ובזכותה ניתן לכתוב קוד ב typescript, מחוץ לריצה של דפדפן, שזה בדיוק מה שאנחנו צריכים בשביל הבדיקות שלנו.

הורדה של הסביבה:

<https://nodejs.org/en>

## התקנת Git

Git הוא כלי הניהול שלנו, איתו, אנו שומרים, משתפים ומנהלים את הקוד שלנו, בשרת חיצוני, שנועד להגן לנו על הקוד.

הורדה של git:

<https://git-scm.com/downloads>

## ספריות NPM

נכון לעכשיו, אנחנו **לא נצטרך** את הספריות האלו מותקנות על המחשב שלנו.

אבל, בגלל שאנחנו כן משתמשים בהם במהלך כל ההכשרה של הבדיקות, נכתוב על כל ספריה כמה מילים, ועל איך להתקין ספריות של node למחשב בכלליות.

במידה ואנו רוצים ליצור פרויקט חדש, ולהתקין בו ספריות, אנחנו נצטרך קודם כל, ליצור תיקייה חדשה.

לאחר שיצרנו את התיקייה, נפתח בה CMD, ונקליד את הפקודה הבאה:

npm init

לחצו enter על כל הדברים שהפקודה מציעה לכם, עד שהיא מסיימת.

פקודה זו תיצור לכם את הקובץ package.json.

קובץ זה מכיל לנו את כל המידע הנחוץ של הפרויקט, כמו השם של הפרויקט, גרסת הפרויקט, ספריות הפרויקט, סקריפטים להרצה ועוד.

כל ספריות הפרויקט יהיו תחת האובייקט dependencies, עם השם שלהם ומספר הגרסה שלהם.

לאחר שיש לנו את הקובץ, נוכל להתחיל להוריד ספריות.

לפני כל שם של ספרייה שמוצג לנו למטה, נקליד את הפקודה הבאה:

npm i

פקודה זאת מתקינה כל ספרייה שבאה אחריה.

NPM היא מערכת ניהול חבילות (ספריות) שבאה כברירת מחדל עם ההתקנה של Node.JS.

האות i היא בשביל install.

למען הדוגמא, צרו תיקייה בשולחן העבודה, פתחו בה CMD, והעתיקו את השם של אחת מהספריות יחד עם הפקודה npm i.

**וודאו** שהיא נוספה לקובץ package.json

וגם, וודאו שנוצר לנו תיקייה חדשה שנקראת node\_modules. בתיקייה זו, יש לנו את כל הספריות שאנחנו מורידים, הם למעשה נשמרים לוקלית בתיקייה של הפרויקט.

נקודה למחשבה

נגיד שעכשיו אנחנו מעתיקים פרויקט קיים למחשב שלנו, וכשאנחנו רוצים להריץ אותו הוא מראה לנו שחסרים לנו הספריות של הפרויקט.

אבל, אם ניכנס לקובץ package.json, נראה שיש לנו את כל השמות של הספריות יחד עם המספרי גרסה שלהם.

אז, מה הבעיה? ואיך אפשר לפתור אותה?

חישבו והחזירו תשובה לחופף שלכם.

### רשימת הספריות

#### ספריית babel

@babel/cli

@babel/core

@babel/preset-env

@babel/preset-typescript

##### הסבר קצר על הספרייה

אנו כותבים קוד בגרסה הכי חדישה של JS.

אולם, כאשר המחשב שלנו מקמפל את הקוד לשפת מכונה, הוא לא מכיר את הפקודות החדשות של - JS יש להציע.

בשביל זה, נועדה הספרייה babel, עם כל התלויות שלה.

ספרייה זו ממירה קוד JS עדכני, משנת 2015 ומעלה, ל JS יותר ישן

סיבה נוספת שאנו משתמשים ב babel, היא בגלל ש cucumber-js לא תומך בגרסאות החדשות של ES6 ו typescript…

בהמשך ההכשרה, נראה איפה זה בא לידי ביטוי.

#### ספריות cucumber

@cucumber/cucumber

cucumber-html-reporter

##### הסבר קצר על הספריות

הספרייה הראשונה היא בשביל לכתוב קוד ב cucumber, זהו ה - npm הכי עדכני של cucumber.

השנייה, היא בשביל דוח בדיקות.

למעשה, אחרי כל הרצה של הבדיקות, אנו רוצים שהאנשים, בין אם זה בוס, מפקד, מפתחים, יבינו בסופו של יום איזה בדיקה לא עברה, ומה הסיבה שהבדיקה לא עברה.

ולכן, ל cucumber יש ספרייה ממש נוחה, שמייצא דוח בדיקות, עם כל תכולה שאנו כותבים, כל תרחיש, ובמידה והתרחיש נכשל, איפה הוא נכשל, ומה סיבת הכשל, עם תמונה של המערכת.

#### ספריית playwright

@playwright/test

playwright

#### ספריות אחרות

dotenv

ts-node

typescript

##### הסבר קצר על הספריות

הספרייה dotenv לוקחת ערך של משתנים מתוך קבצי .env

בדרך כלל, אנו נכניס לקבצים אלו משתנים שהם קבועים, כמו הכתובת של המערכת שאנו פותחים, הגדרות של המערכת, נתיב של הקבצים השימושיים שהמערכת צריכה, וכדומה.

הספרייה ts-node ממירה קוד TS לקוד JS מבלי לבצע קומפילציה מקדימה, בעזרת Node.JS.

# Git

## מבוא

כפי שכתבנו בסעיף ההתקנות, git היא תוכנה שנועדה לשמור, לנהל, ולשתף את הקוד שאנו כותבים.

כאשר אנו שומרים את הקוד שלנו, הוא למעשה נשמר בשרת חיצוני, זה יכול להיות gitlab, github, azure devops.

על מנת לא לספיילר יותר מדי על הכלי הנהדר הזה, נעבור להכשרה.

## הכשרה

<https://learngitbranching.js.org/>

כאן, יש תרגילים שנועדו להראות לנו איך מתעסקים עם git

לרשותכם שבוע לבצע את כל התרגילים

בהצלחה 😊

# GitHub

## מבוא

בסופו של דבר, אנו נצטרך לעבוד בצוות, ועבודה בצוות, חוץ משיתוף פעולה, דורשת שיתוף של הקוד שלנו עם כל הצוות.

בשביל זה, נועדו שרתים מרוחקים, שנועדו לשמור את הקוד שלנו במקום בטוח

ובשביל זה, נועד גם הגיט.

כל הפקודות שלמדתם עכשיו בגיט נועדו ללמד אתכם איך לשתף את הקוד, ואיך להתנהל נכון עם קוד.

כרגע, אנו נשתמש באתר הנקרא GitHub, שהוא אחד מיני רבים לשמירת ותיעוד קוד נכון.

אז בואו נתחיל 😊

## הכנת סביבת העבודה

היכנסו לקישור הבא:

<https://github.com/>

צרו משתמש חדש באתר.

על שולחן העבודה שלכם, פתחו git bash, והקלידו את הפקודה הבאה, עם הקישור הבא:

git clone https://github.com/robin8360/General-QA-Training.git

זה יעתיק את הפרויקט General-QA-Training למחשב שלכם, בתוך תיקייה שתקרא באותו השם של הפרויקט.

אם אין לכם הרשאות ל repository הזה, גשו לחופף שלכם שיעניק לכם הרשאות.

לאחר מכן, אתם מוכנים להתחיל לעבוד כמו מקצועניים.

אז מה עושים?

על כל תרגיל שאתם עובדים מהיום והלאה, אתם יוצרים ענף חדש

עם הפקודה המוכרת שלמדתם מההכשרה של גיט:

git branch <branch-name>

או (במידה ואתם רוצים לפנות לענף זה ישירות):

git checkout -b <branch-name>

לאחר מכן, אתם עובדים על התיקייה כרגיל.

במידה ועשיתם המון שינויים, אתם מתחילים ליצור קומיטים, כל קומיט מסביר מה כתבתם עד עכשיו בקוד.

שימו ❤ חשוב ליצור כמות מכובדת של קומיטים בקוד, כיוון שככה נוכל לעקוב כמו שצריך אחרי שינויים שעשינו. כמובן, שזה לא אומר עכשיו ליצור קומיט לכל קובץ, אלא, על כל פונקציונליות כללית.

## דחיפת השינויים לשרת המרוחק

צרו ענף חדש, וצרו בתיקייה של הפרויקט קובץ טקסט

כתבו בקובץ טקסט רשימה של 5 מאכלים שאתם אוהבים, ו 5 חיות שאתם הכי אוהבים.

לאחר מכן, כתבו ב git bash את הפקודה הבאה:

git gui &

זה יפתח לכם חלון חדש, זהו ה git gui

בואו נעבור על כל חלק של החלון שנפתח:

מצד שמאל למעלה, יש את כל הקבצים ששינינו, אבל שעדיין לא הוספנו לקומיט, הם יהיו ב unstaged changes.

אחרי שנלחץ על האייקון בצד שמאל של הקובץ, הוא יכנס ל staged changes, כלומר, שהוא מוכן להיות חלק מהקומיט שאנחנו יוצרים.

אם אתם רוצים לראות את השינויים שעשיתם בקובץ, אתם לוחצים על השם של הקובץ, והתוכן שלו יהיה מוצג בחלק הימני של החלון, עם כל השינויים שביצענו.

בחלק התחתון, אתם כותבים את ההודעה של הקומיט (commit message), על ההודעה להיות מפורטת, אך ברמה המתאימה, לפי מה שאתם עושים לו קומיט.

אחרי שסיימתם, אתם לוחצים על הכפתור commit, וזה מבצע את הקומיט.

אולם, השינויים עדיין לא נשמרו בשרת המרוחק, כי עדיין לא דחפתם את השינויים.

בשביל זה, אתם לוחצים על כפתור ה push, לוחצים push בחלון שנפתח, וזה ידחוף את השינויים שלכם לשרת המרוחק.

מעולה, דחפתם את השינויים שלכם לשרת המרוחק, ב GitHub, יש כפתור שבו אפשר לראות רשימה של כל ה הענפים הקיימים, שם יהיה את הענף שיצרתם, ואם תיכנסו אליו תראו את הקובץ שיצרתם.

## Pull Request

במידה וסיימתם את התרגיל ואתם רוצים שהוא יהיה חלק מהקוד הכולל של כל הצוות, אתם יוצרים pull request.

אחרי שהקוד שלכם עובר סקר קוד, הוא נכנס לקוד של הפרויקט הכולל, והענף שלכם מתמרג'ג' (merge) עם הענף הראשי, ה main.

פתיחת Pull Request ב GitHub:

היכנסו לפרויקט באתר, לחצו על כפתור Pull Request בחלק העליון של הפרויקט.

לחצו על הכפתור New Pull Request, זה יפתח בדף חדש את אפשרות הבחירה לבחור איזה ענף אתה רוצה למרג'ג' עם איזה ענף.

לאחר מכן, אתה כותב כותרת ל PR, תיאור מפורט של מה יש ב PR, ואז לוחץ על יצירת ה PR.

לאחר שתעבור סקר קוד, תוכל לראות את ההערות שהחופף שלך כתב תחת הלשונית Files changed.

## Git Rebase

בגלל שאנחנו אנשי גיט מנוסים, לפני שאנחנו נדחוף את הענף שלנו לענף הראשי (במידה ועברתם את הסקר קוד), אנחנו נזכור תמיד לעשות pull ל main, כדי לקבל את כל השינויים במידה ויש שינויים, ונשתמש בפקודה המופלאה:

git rebase main

זה מעדכן לנו את הענף הנוכחי, עם הענף הראשי, על ידי לקיחת כל הקומיטים שביצענו, והכנסתם מעל הענף הראשי.

בשונה מ merge, שלוקח את כל הקומיטים שעשינו, ומאחד אותם לקומיט אחד (פחות קריא, יוצר סיבוך בעץ גיט).

במידה ויש קונפליקטים בקוד, נצטרך לדעת איך לפתור אותם, השתדלו להיות ערניים למה למחוק ומה לא למחוק, כיוון שיכול להיות שיש שינויים שחברי הצוות עשו שנוגע בקבצים שאתה נגעת בהם, ולכן, יש לתת לזה תשומת לב.

זהו, זה היה הסבר קצר על להעלות את הקוד שלנו לשרת מרוחק.

כל שנותר לנו עכשיו זה להתחיל לכתוב קוד, ולהשתמש בגיט בצורה חכמה.

אז קדימה לעבודה 😊

# שפת typescript

## מבוא

השפה בה אנו נכתוב נלמד לכתוב בדיקות, הינה שפת typescript

שפה זו משומשת בעיקר לכתיבת אתרים, בזכות הדינמיות שלה, ומשתמשת בסביבת ההרצה node.js

אנו משתמשים בשפה זאת בשל הנוחיות של הבודקים בצוותינו להנגיש אותה למפתחים.

לרשותכם שבועיים לקרוא את כל החומר, ולענות על כל התרגילים.

בהצלחה 😊

## הכשרה

השתמשו בתיקייה שפתחתם בה שרת לגיט (General-QA-Training), כיוון שאתם לא צריכים את כל הספריות שהצגנו בסעיף ההתקנות.

בתיקייה זו, אתם צריכים את הדברים הבאים:

ליצור קובץ package.json עם הפקודה מתחילת ההכשרה

ליצור ידנית קובץ index.ts – שבו תכתבו את הקוד

ליצור קובץ tsconfig.json – על ידי הרצת הפקודה הבאה:

tsc --init

שימו לב, בקובץ tsconfig, תצטרכו להוריד מהערה את הערך הבא:

SourceMap: true

זאת על מנת שתוכלו לדבג את הקוד שלכם.

בקובץ package.json, יש את האובייקט scripts, ובתוכו יש סקריפט שנקרא test

סקריפטים הם דרך נוחה של NPM, להריץ את הקוד שלנו מבלי לכתוב כל פעם את הפקודה להרצה.

שנו את השם של test ל start, וכתבו את הסקריפט הבא בתוך המרכאות:

tsc && node index.js

זה ימיר את כל הקבצי ts לקבצי js, ויריץ את הקובץ js, שהוא כמו ה main של המערכת.

### חומר קריאה

היכנסו לקישור הבא:

<https://www.typescripttutorial.net/>

כאן יש את כל החומר הנלמד על שפת typescript, פתחו מחברת, וכתבו לעצמכם את כל החומר הנלמד, בצורה שרק אתם תוכלו להבין הכי טוב.

### תרגולים

1. היכנסו לקישור הבא:

<https://www.w3schools.com/typescript/typescript_exercises.php>

בצעו את כל התרגילים, השתדלו לענות לפני שאתם מוודאים את התשובה.

1. היכנסו לקישור הבא:

<https://www.codecademy.com/learn/learn-typescript?utm_source=ccblog&utm_medium=ccblog&utm_campaign=ccblog&utm_content=cw_javascript_coding_challenges_beginners_blog>

כאן אנו נלמד על כל חלק של ts בנפרד, באמצעות תרגילים ממש נחמדים

היכנסו עם משתמש google, בנתיים, בצעו רק את החומר הלימודי, תקראו לחופף שלכם כשסיימתם הכל.

1. היכנסו לקישור הבא:

<https://typescript-exercises.github.io/#exercise=1&file=%2Findex.ts>

לפניכם 16 תרגילים, כאשר בכל תרגיל, יש שגיאת קומפילציה כזאת או אחרת.

פתרו את כל התרגילים.

1. לפניכם 3 תרגילים לפי סדרי קושי: קל, בינוני, קשה

קל:

<https://exercism.org/tracks/typescript/exercises/pangram>

בינוני:

<https://exercism.org/tracks/typescript/exercises/binary-search>

קשה:

<https://exercism.org/tracks/typescript/exercises/knapsack>

בצעו את כל התרגילים.

אחרי כל תרגיל שאתם מסיימים, שלחו את הקוד שכתבתם לחופף שלכם, על מנת לבצע code review.

שימו לב שכל תרגיל כזה צריך להיות בפונקציה נפרדת.

כל הכבוד, סיימתם את הכשרת typescript 😁

גשו לחלק הבא

# Postman ובקשות API

## מבוא

לפני שנתחיל להתעסק באוטומציה, נצטרך ללמוד עוד כלי אחד חמוד ומאוד פשוט, הוא כלי ה postman.

כלי זה נועד לשלוח בקשות API ישירות לשרת של האתר, מבלי להתעסק ב UI שלו.

אבל נראה לי שקפצנו מהר מדי, מה זה בכלל בקשת API?

שמחה ששאלתם, כי בחלק זה נכסה נושא זה, וגם ניגע בכלי שאיתו נבצע בקשות API.

אז קדימה, 3, 2, 1

יצאנו לדרך!

### בקשות API

אנחנו, המשתמשים, נכנסים לאתר כלשהו, למשל, google.

ב google יש לנו את התצוגה הרגילה של האתר, עם הלוגו של האתר, שדה הכתיבה, כפתור חיפוש ועוד.

אבל משהו שאנחנו לא מודעים אליו שקורה מאחורי הקלעים, הוא שרק בטעינה של הדף, מתבצעות המון בקשות API.

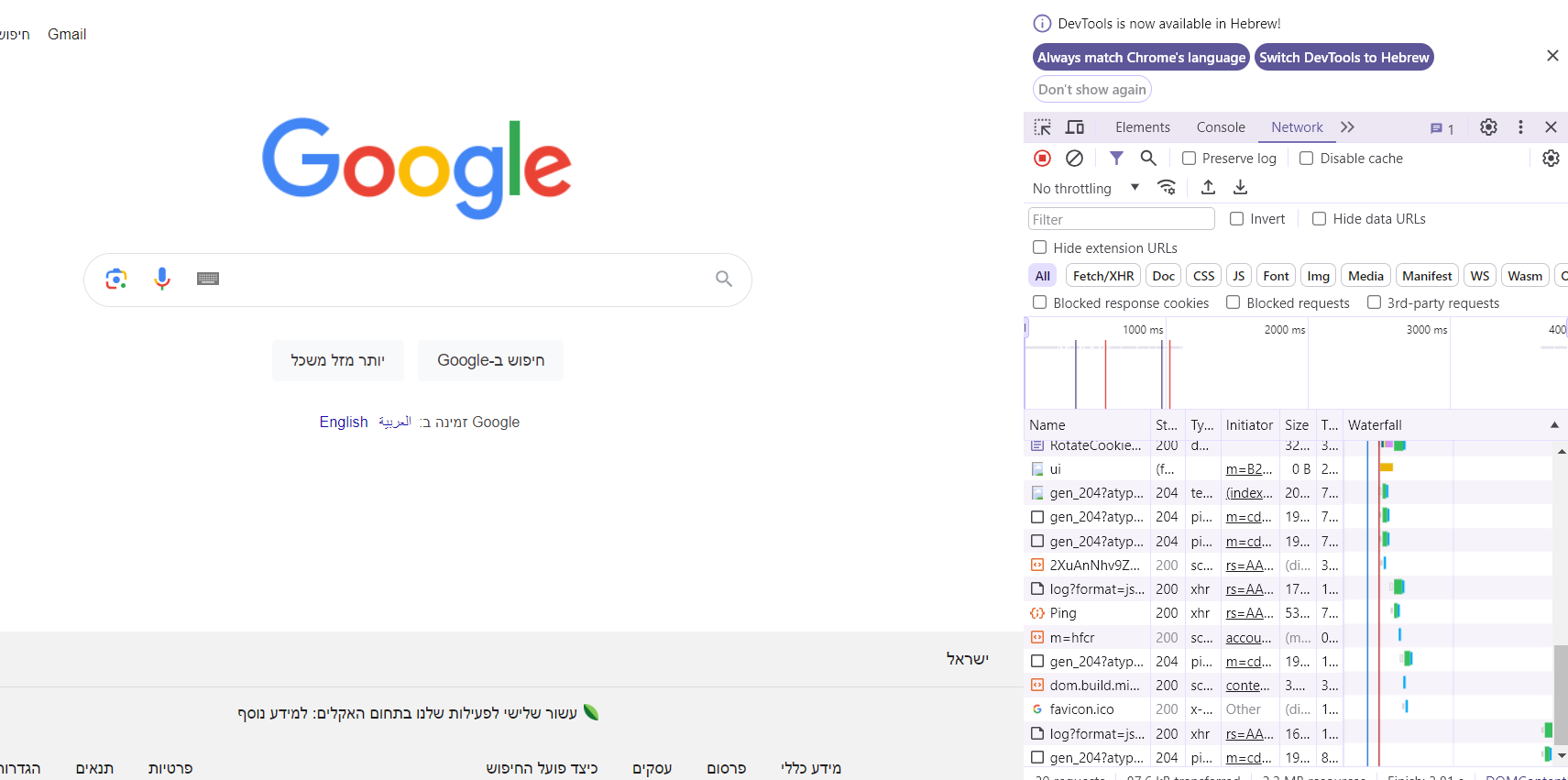
**בקשת API** היא למעשה בקשה שהאתר (הדף) מבצע, על מנת לקבל מידע מהשרת שאיתו הוא מתקשר.

במקרה שלנו, אתר google מבקש בעת עליה שלו את הלוגו, ואת הדף שנטען עם כל הכפתורים והשדות, ועוד דברים שאנחנו לאו דווקא מודעים אליהם או רואים אותם ויזואלית.

בואו נראה את התעבורה של הבקשות שהדף שולח בצורה ויזואלית, פיתחו את אתר גוגל, ולחצו על מקשי המקלדת CTRL+SHIFT+I, או F12

זה יפתח את חלונית ה inspector, שדרכה אפשר לראות את האלמנטים של הדף, את התעבורה של הרשת של הדף, וכדומה.

אם נכנס ללשונית ה network, ונרענן את הדף של גוגל, נראה שהאתר מבקש המון בקשות מהשרת



כל הבקשות שהדף מבקש מהשרת של גוגל

אם נלחץ על אחת מהשורות המוצגות לנו ברשימה, נוכל לראות מידע יותר מפורט על הבקשה, כל בקשה שהדף שולח באה עם תבנית מובנית של מה צריך להיות בתוך הבקשה.

אני לא אפרט יותר מדי, כי אנחנו נכנס כאן לעולם של רשתות, אם אתם רוצים להרחיב על כך, מוזמנים לחפש בגוגל על בקשות ועל מבנה של בקשה.

מה שאנחנו צריכים לדעת, היא שבקשה מכילה מספר דברים:

סוג הבקשה – האם הבקשה היא בשביל לקבל מידע, האם הבקשה היא בשביל לעדכן משהו בדף, האם הבקשה היא בשביל למחוק משהו בדף, וכדומה.

סוגי הבקשות:

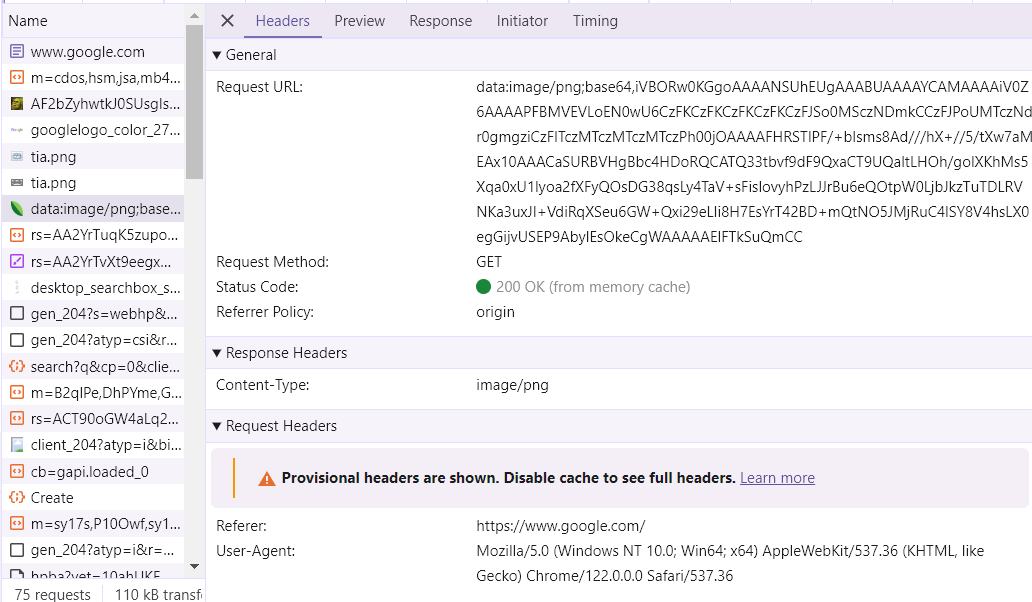
* GET – בקשה של קבלת מידע
* POST – בקשה לעדכן מידע בדף
* DELETE – בקשה למחוק מידע בדף
* ועוד...

URL של השרת – בסוף הדף פונה לשרת שממנו הוא רוצה לקבל את המידע, לכן צריך לציין את הכתובת של השרת, כדי שהוא יקבל את הבקשה ויחזיר תשובה בהתאם.

ראש הבקשה – בחלק מהמקרים, נצטרך להוסיף מידע לראש (header) של הבקשה, אנו נעשה זאת בעיקר בשביל הגוף של הבקשה, האם הבקשה היא מסוג json, מה הגודל של גוף הבקשה וכדומה.

גוף הבקשה – בחלק מהבקשות, אנו נצטרך להעביר מידע מסוים כחלק מהבקשה עצמה, על מנת לקבל מידע יותר מקיף מהשרת, או כדי לעדכן את הדף עצמו.

בואו ניקח דוגמא מבקשה בגוגל (ניקח בקשה יחסית קטנה כי רוב הבקשות בגוגל ארוכות ומעצבנות, בהמשך נעבור לאתר יותר ידידותי למשתמש)



3

1

2

4

כאן אנחנו רואים מספר דברים:

1. הכתובת של הבקשה – בדרך כלל הכתובת תהיה יותר יפה, פשוט בגלל שזה גוגל היא נראית ככה

2. סוג הבקשה – הבקשה הייתה רק בשביל לקבל מידע, לכן GET

3. סטטוס הבקשה – מה שהשרת החזיר, 200OK אומר שהשרת קיבל את הבקשה בהצלחה

4. סוג גוף הבקשה – התשובה שהשרת החזיר במקרה כאן הוא מסוג תמונה (image/png)

שימו ❤ - אין כאן גוף של בקשה, בדרך כלל בקשות GET לא יהיו מלוות בגוף, אלא רק בכתובת של השרת, זאת כיוון שכשאתה מצפה לקבל מידע, אתה לא רוצה להעביר מידע בעצמך, אלא רק אם אתה מעדכן מידע בדף אתה צריך להעביר מידע אודות הדבר שאותו אתה מעדכן.

אוקי, אחרי שנגענו קצת בבקשות API, בואו נעבור לתוכנת ה Postman

### תוכנת ה Postman

אוקי, עכשיו אחרי שהתעסקנו קצת עם גוגל, בואו נזניח אותו ונעבור לאתר יותר ידידותי למשתמש.

דבר ראשון, התקינו את תוכנת ה postman

<https://www.postman.com/downloads/>

היכנסו לקישור הבא:

<https://github.com/alicevillar/restful_api_exercise>

זהו פרויקט ממש חמוד של API, שאתם מריצים אותו על המחשב שלכם ואתם יכולים לשלוח בקשות API

בצעו clone של הפרויקט על שולחן העבודה שלכם, ולאחר מכן נבצע מספר פעולות על מנת לגרום לו לעבוד, אז נפתח לזה כותרת חדשה 😱

#### הפעלת מערכת ה API

דבר ראשון שנעשה זה להוריד פייתון, במידה ולא קיים לכם על המחשב, היכנסו לקישור הבא:

<https://www.python.org/downloads/>

ממליצה לבחור את פייתון גרסה 3.8

לאחר מכן, גשו למשתני הסביבה שעל המחשב שלכם, והוסיפו את הנתיב של פייתון שיצרתם למשתני הסביבה על המחשב שלכם. במידה ועוד לא יצא לכם להתעסק עם משתני סביבה, ממליצה לחפש באינטרנט add environment variable to path python

עכשיו, אחרי שיש לכם גם פייתון על המחשב וגם את הפרויקט מה repo, נפתח cmd בתיקייה של הפרויקט, ונכתוב את הפקודות הבאות:

pip install flask

pip install flask\_restful

לאחר מכן, נכתוב את הפקודה הבאה:

python ./main.py

וזה יריץ את הפרויקט תחת הנתיב הבא:

<http://127.0.0.1:5000>

#### נמשיך את הכשרת postman

בואו נעשה דוגמא קטנה, לצד הכתובת שכתבתי כאן, תוסיפו את המילה /user/all בשדה העליון.

זה יביא לכם אובייקט מסוג json עם כל המשתמשים שיש באותו אתר.



אוקי, עכשיו, זה נחמד והכל לכתוב דרך הדפדפן את הבקשות, אבל הבעיה היא שאי אפשר לכתוב בקשות POST ובקשות אחרות שמשנות את האתר דרך הדפדפן, אלא רק בקשות GET.

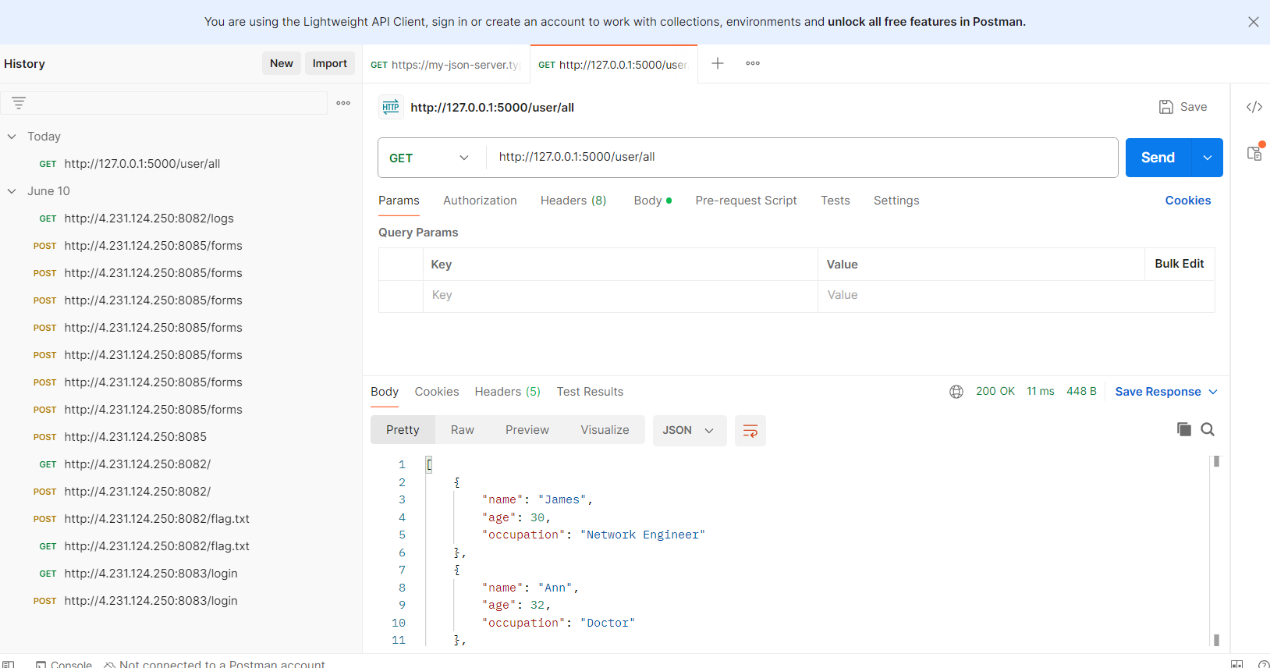
אז, בואו נפתח את התוכנה החמודה שלנו שנקראת Postman, ונתחיל לתפעל אותה 😆

אז קודם כל, בואו נשלח בקשת GET פשוטה.

לחצו על יצירת בקשה חדשה (כפתור ה +), הוסיפו את ה URL לשדה, וודאו שהבקשה היא מסוג GET, ותלחצו על send

זהו! סה"כ זה די פשוט

תוכלו לראות את התשובה של הבקשה למטה, עם סטטוס הבקשה 200OK ואת המידע שהשרת מחזיר.



כתובת הבקשה

סוג הבקשה

שליחת הבקשה

מידע מהשרת

סטטוס הבקשה מהשרת

## סיכום קצר

בואו נעשה recap על מה שלמדנו כאן

עד עכשיו, למדנו על בקשות API, ועל תוכנת ה postman

אבל לא ממש הוסבר למה אנחנו לומדים את זה, מה הפואנטה? בסוף ההכשרה הזאת נועדה בשביל לכתוב אוטומציה למערכת תצוגתית.

אז כן ולא, ונסביר לכם למה.

בעיקרון, כן, עיקר הבדיקות שלנו הולכות להיות תצוגתיות, שזה אומר ללחוץ על דברים, להזין תווים בשדות, דברים בסגנון הזה.

אבל, לא, מכיוון שיש לנו דברים שאנחנו לא יכולים לעשות בלי בקשות API. בסופו של יום, אנחנו לא כותבים בדיקות למערכת פשוטה כמו אתר Ynet שהוא פשוט נועד להציג לנו כתבות. אנחנו כותבים בדיקות למערכת דינמית שדברים משתנים בה, ובמצב של בדיקות מעבדה, בלי חיבור של ממשקים אמיתיים, אנחנו נצטרך לעבוד עם סימולטורים שעיקר העבודה איתם היא בקשות API.

אז בפועל, יש המון דברים שמתבצעים בבקשות API, אני לא ארחיב אותם כאן מטעמי ב"מ, אבל בהמשך הדרך אתם תראו.

## תרגול

חקרו עוד קצת על postman, על איך שולחים בקשות מסובכות יותר, איך מעדכנים פרטים מאתר ואיך מוחקים, וכדומה.

הכנסו לקישור שהצגנו בהסבר:

[http://127.0.0.1:5000/](http://127.0.0.1:5000/user/all)

בצעו את הבקשות הבאות, וקראו לחופף שלכם להסתכל על הבקשות דרך ה postman.

### בקשות

על כל בקשה שאתם עושים, וודאו שהמשתמש שיצרתם, מחקתם, עדכנתם וכדומה, ברשימה של כל המשתמשים.

בנוסף לכך, שימו לב שאת רוב הבקשות אתם מבצעים תחת הכתובת:

http://127.0.0.1:5000/user/

1. בקשה ליצירת user חדש
2. בקשה ליצירת user קיים
3. בקשה לעדכון user קיים – ודאו ברשימה הגדולה
4. מחיקת user – וודאו שהוא לא קיים יותר ברשימה
5. הצגת מידע של user אחד מהרשימה

# אלמנטים באתר האינטרנט

## מבוא

לפני שנתחיל ללמוד על אוטומציות, ועל איך לכתוב קוד מטורף שלוחץ לנו על כפתורים, בוא נתחיל מקטן.

לרשותם יום לעבור על חלק זה, ולסיים תרגילים אלו

בהצלחה😊

### חלונית ה inspect

בסופו של דבר, כשאנו כותבים בדיקות שמוודאות תצוגה של מערכת, אנו מסתמכים על האלמנטים שיש למערכת.

שזה למעשה, כל מה שהאתר מכיל.

למשל, אם ניכנס ל – google, נראה שיש לו כמה אלמנטים עיקריים:

לוגו, תיבת טקסט, שני כפתורים, ועוד המון אלמנטים.

בואו נפתח את חלונית ה inspect, חלונית זו נועדה לתת מענה מלא על הדף שעליו אנו נמצאים, האלמנטים שיש בו, הבקשות שהדף מייצא, שגיאות ריצה במידה וקיימות, וכדומה.

נפתח את החלונית על ידי לחיצה על מקש ה 12F, או על ידי הקומבינציה CTRL+SHIFT+i

בתוך חלונית זו, יש כמה לשוניות.

עברו ללשונית ה Elements, וכאן, ניתן לראות את כל האלמנטים של הדף.

חלק מהאלמנטים הם ממש נראים לעין, כמו כפתור (button), תיבת טקסט (input), וכדומה. בעוד כמה נועדו לשמור על הסדר של הדף, על מרכוז של אלמנטים (div, center), וכדומה.

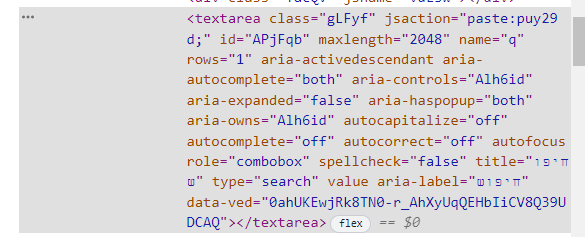
ליד השמות של הלשוניות, יש לנו כפתור קטן שנראה כמו עכבר שלוחץ על מסך:



לחצו עליו, ולאחר מכן, תראו שכשאתם גוררים את העכבר לעבר הדף עצמו, הוא מסמן כל אלמנט (גם אלמנטים שלא רואים) ומציג אותו בחלונית.

עברו עם העכבר על שורת החיפוש של גוגל, בעוד שאתם במצב של inspect, ולחצו עליה.

זה מה שיהיה מוצג:



זהו אלמנט מסוג textarea, ויש לו כמה פרמטרים, כל פרמטר כזה נקרא attribute.

בעזרת attributes אלו, ניתן לחפש את האלמנטים באוטומציה.

עכשיו אתם בטח שואלים, מה הקשר? אז בשביל זה אנחנו כאן.

בסוף, האוטומציה מחפשת את האלמנטים שעליה היא לוחצת, מזינה טקסט וכדומה, דרך דבר שנקרא זיהוי, לכל אלמנט בדף יש זיהוי, יש אלמנטים שיש להם את אותו הזיהוי, ויש אלמנטים שיש להם זיהוי ייחודי.

אנחנו תמיד נשאף למצוא זיהוי ייחודי לכל אלמנט, כדי לא להתחיל לחפש אלמנטים בצורה שיטתית ולא יציבה.

לא יציבה משמע, שכאשר נריץ את אותה הבדיקה פעמיים, במקרה אחד הבדיקה תמצא את האלמנט, ובמקרה אחר, היא לא תמצא, כי הוא השתנה, או שהוא לא ייחודי מספיק.

אז בואו נמצא את הזיהוי של תיבת הטקסט.

### מציאת זיהוי ייחודי

על מנת לחפש זיהויים בדף, לחצו על לשונית ה console.

בלשונית זאת ניתן לבצע מספר פעולות, ניתן לאסוף מידע על האתר, על ידי כתיבת פקודות ב JavaScript, ובנוסף לכך, מוצגים לנו לוגים של שגיאות, במידה ויש שגיאות.

בואו נחפש את האלמנט דרך ה tag name שלו, שזה המילה textarea. כתבו את הדבר הבא לתוך שורת הטקסט:

$$("textarea")

לחצו enter, ותראו שהאתר מוצא לנו יותר מאלמנט אחד של textarea, משמע, הזיהוי לא יציב.

בואו ננסה לפי class, כתבו את הדבר הבא לתוך שורת הטקסט:

$$(".gLFyf")

ניתן לראות כי כאן, אנחנו מקבלים רק אלמנט אחד, משמע, הוא יחסית יציב.

אולם, יש בעיה עם class, ניתן להשתמש בו בכמה מקומות. class נועד בעיקר לקחת מקבץ של אלמנטים, כמו רשימה, ולתת להם זיהוי אחד, זאת על מנת להקל על המפתחים כשהם יוצרים רשימה באתר, וצריכים לעצב כל אלמנט ברשימה.

נ.ב: בגוגל, חלק מהאלמנטים יהיו עם class או id דינמיים, משמע, יכול להיות שיום אחד ה class יהיה gLFyf, ויום אחר, ה class יהיה sHE21r, לכן, צריך לקחת אלמנטים של גוגל בעירבון מוגבל.

אצלנו במערכת, הסיכוי שאלמנטים ישנו את ה attributes שלהם הוא אפסי.

בואו ניקח את הזיהוי הכי יציב, שהוא ה id, id הוא ייחודי לכל אלמנט, לא יהיו שני אלמנטים עם אותו ה id. אולם, בגלל שאמרנו ש google מעצבן עם הזיהויים שלו, אז לא ניקח אותו כרגע בחשבון, בכל מקרה, אם נזין ב console את הקטע הבא:

$$("#APjFqb")

ניתן לראות כי הוא מוצא את האלמנט בלי בעיות.

בואו ניקח זיהוי, שמאז שאני מכירה את עצמי כותבת אוטומציות עדיין קיים שם, הוא ה name. כתבו את הדבר הבא ל console:

$$("[name='q']")

זהו הזיהוי כרגע הכי יציב שיש לאלמנט הנוכחי.

למרבה הצער, לא תמיד יהיו לנו זיהויים ייחודים, זה בעיקר בגלל חוסר תשומת הלב של המפתחים, כשהם כותבים את המערכת.

אצלנו במערכת, שמנו attribute שנקרא data-id, וכל אלמנט הוא עם שם ייחודי. לכן, אם אתם רוצים לקדם אצלכם כתיבת קוד נכונה בצוות, שווה לכם לדבר עם ראש הצוות או המפתחים, ולבקש מהם להוסיף לכם זיהוי ייחודי לכל אלמנט.

עכשיו, אחרי שלמדנו קצת על איך לחפש אלמנטים בדף, גשו להכשרה.

## הכשרה

היכנסו לקישור הבא:

<https://flukeout.github.io/>

כאן יש תרגילים ממש נחמדים על css, ועל איך לחפש אלמנטים באינטרנט בצורה מקצועית.

בהצלחה 😊

# Design patterns בבדיקות

לפני שנתחיל את ההכשרה על אמת, ונכתוב בדיקות כמו בודקים מקצועניים, יש נושא אחד אחרון שחשוב לדבר עליו, והוא תבניות העיצוב בבדיקות.

אם אני אקח את ההגדרה המלאה מויקיפדיה, ואכניס אותה לכאן:

"תבנית עיצוב הוא תיאור של פתרון כללי לבעיה שכיחה בעיצוב תוכנה. תבניות עיצוב איננה פתרון מדויק שניתן להעבירו הישר לקוד, אלא היא תיאור דרך לפתרון בעיה, שעשויה להיות שימושית במצבים רבים."

עכשיו, תתארו לעצמכם שאתם עובדים במפעל לאריזת משלוחים.

אתם מקבלים קופסא, סרט, דבק, פצפצים, ואת המוצר שאותו אתם צריכים לעטוף.

אם היו אומרים לכם: "פשוט תעטפו את זה איכשהו ותשלחו"

הייתי יכולה לחשוב על כמה דרכים לעטוף:

1. להכניס את המוצר ישירות לקופסא, לא לסגור אותו ולא לעטוף
2. לעטוף את המוצר במלא דבק סלוטייפ ולשלוח ככה
3. לשים בקופסא פצפצים, לסגור את הקופסא עם הדבק, לשים את המוצר מעל הקופסא
4. לשים את המוצר בתוך הקופסא, ולעטוף את הקופסא עם פצפצים
5. לשים את הפצפצים ואת הדבק בתוך הקופסא ולשלוח אותה בלי המוצר

בקיצור, יש לנו כאן המון דרכים לשלוח מוצר ולעטוף אותו😅

אבל אם היו נותנים לכם תבנית של מה לעשות: "לעטוף את המוצר עם פצפצים (כשהמוצר נמצא בתוך הפצפצים כמובן), להוסיף על הפצפצים קצת דבק כדי שהפצפצים לא יפתחו, לשים את המוצר בקופסא ולסגור את הקופסא עם סרט"

הייתם עושים בדיוק את התבנית הזאת, על כל המוצרים שהיו נותנים לכם.

שימו לב גם, שלא דיברו פה על איזה מוצר לעטוף, כל מוצר שמביאים לכם אתם תעשו ככה.

אז איך זה מתקשר אלינו?

נניח שעכשיו אתם רוצים לבנות פרויקט בתכנות, אם תתחילו לכתוב את הקוד כמו איך שאמרו לנו לעטוף מוצר בפעם הראשונה, נגיע לקוד מאוד מבולגן, לא יעיל, ולא טוב לתחזוקה.

ובשביל זה, נועדו תבניות העיצוב.

תבניות העיצוב בעצם מקנות סט של חוקים, שאומרים איך הקוד שלנו צריך להיראות, איך כל מחלקה שניצור תיראה, עוד לפני שבכלל כתבנו אותה.

צריך להבין שתבניות עיצוב זה גישה תפיסתית ולא תכנותית, היא לא אומרת לנו מה לכתוב בקוד, היא אומר, איך לעצב את הקוד כך שיראה יותר מובן וברור.

בין היתרונות של תבניות עיצוב הן

1. חיסכון בזמן – לא צריך להמציא את הגלגל
2. קוד ברור ומובן – מתכנתים אחרים יזהו את הדפוס
3. תחזוקה קלה יותר – הקוד מאורגן לפי עקרונות ברורים

## תבניות עיצוב בבדיקות

אז איך זה מתקשר אלינו בבדיקות?

בגלל שלפתח אוטומציה זה כתיבת קוד, יש כמה סוגים של תבניות עיצוב שמשתמשים בהם בשביל תשתית בדיקות.

### POM

התבנית הכי נפוצה, שבטח עסקתם בה בקורס, היא POM (Page Object Model)

תבנית זו נועדה ליצור סדר באלמנטים של המערכת, על ידי כתיבה של מחלקות, שמהוות את הדפים במערכת.

לדוגמא: אם אתם נכנסים לאתר של קניות (למשל Shein), יש לכם כמה דפים בדף:

1. דף המוצרים
2. דף סל הקניות
3. דף של פריט
4. דף כניסה למשתמש
5. דף ביצוע קניה (checkout)
6. דף המשתמש
7. ועוד דפים...

בתבנית זו, אנחנו יוצרים מחלקה כזאת לכל דף, ובכל מחלקה כזו, אנו כותבים את האלמנטים שיש בדף, ומאתחלים אותם.

### POR

תבנית נוספת שהיא מאוד דומה ל POM, היא POR (Page Object Repository)

בתבנית זו, אנחנו מבצעים בדיוק את מה ש POM עושה, כלומר, יוצרים דפים לפי המערכת ומאתחלים שם אלמנטים.

אבל במקום לאתחל את האלמנטים, אנחנו שומרים את השם שלהם כמחרוזת, עם הזיהוי של האלמנט, ואז קוראים לאלמנטים דרך השם שלהם.

ואני בטוחה שאתם בטוח בטוח שואלים את עצמכם, אבל אם יש לי את POM, למה שאני אשתמש ב POR?

קודם כל, שאלה טובה

דבר שני, על אף ש POM היא תבנית באמת חזקה לכתיבת בדיקות אוטומטיות, היא בעייתית עם חזרתיות של פעולות.

בשביל להסביר לכם יותר לעומק על זה, אתם תצטרכו להתחיל את ההכשרת אוטומציה

## הערת צד אחרונה לפני שנתחיל

את **ההכשרה הזאת**, כתבנו בשימוש עם תבנית העיצוב POR

לכן, אם אתם יודעים שבצוות שלכם לא מממשים ב POR אלא בתבנית אחרת, זה עדיין בסדר לעשות את ההכשרה עם התבנית הזאת, מכיוון שהיא דומה להמון תבניות בבדיקות שעובדות באופן הזה.

# Playwright

## לפני לפני שנתחיל – דף קונבנציות

אחרי שתסיימו את כל חומר הקריאה, תתחילו לכתוב את הקוד

חלק זה נועד להוות מסמך קונבנציות שאנחנו מממשים

ולכן, אם אתה או את, בודקים מצוות אחר, עדיין תצטרכו במהלך ההכשרה לכתוב קוד באופן שכתוב פה, ולאחר מכן, תלוי בצוות שאליו תגיעו.

### קונבנציות להכשרה – קבצי TS

1. אין לכתוב נקודה פסיק אחרי כל שורה ;
2. על כל משתנה שניצור, כותבים את הטיפוס שלו

דוגמא:

element: Locator

1. כללי רווחים:
   1. משתנה: טיפוס = אתחול-משתנה
   2. פונקציה (משתנה: טיפוס, משתנה2: טיפוס)
2. חייב לתעד כל פונקציה שנכתוב, ככה שתהיה הכי מוסברת שאפשר

דוגמא לתיעוד נכון של פונקציה:

/\*\*

 \* We go to the config/mappings folder, and from the elementKey we want

 \* We take the element identifier - which is the element from the web

 \* It does it by taking the URL we are in, converting it to the name of the json file

 \* From the config/pages.json file

 \* Searching for the elementKey we give in the function, and extracting the element identifier

 \* We do this to make our code cleaner, elementKey is a way for writing cleaner names

 \* @param page

 \* @param elementKey

 \* @param globalConfig

 \* @returns elementIdentifier

 \*/

export const getElementIdentifier = async (

    page: Page,

    elementKey: ElementKey,

    globalConfig: GlobalConfig

): Promise<string> => {

1. יהיו שתי שורות רווח בין imports לקוד עצמו
2. כל פונקציה שניצור מחזירה ערך, גם אם הוא מטיפוס void
3. כל פונקציה שנכתוב תהיה מהסוג של פונקציית חץ (arrow function)
4. פונקציות צריכות להיות כמה שיותר גנריות, כלומר, לא ליצור פונקציה שעושה פעולה על אלמנט אחד, אלא פונקציה שיכולה להיות שמישה לכמה אלמנטים.

דוגמא טובה לכך היא פונקציה שלוחצת על כפתור, אנחנו לא נרצה ללחוץ על כל כפתור עם פונקציה נפרדת, אלא לקבל את הזיהוי של כפתור במערכת כפרמטר לפונקציה, וללחוץ עליו

1. אם שורה עולה על 90 מילים, יש לרווח ולהוריד שורה איפה שאפשר
2. תמיד נכתוב שמות משמעותיים למשתנים ולפונקציות, גם אם הם יצאו שמות ארוכים
3. בין כל פונקציה לפונקציה תהיה שורת רווח
4. מחרוזת עם ערכים בפנים תכתב באופן הבא

`hello ${name}`

### קראו אותי כשתתחילו לכתוב קוד

אחד מהכלים הטובים לשמור על קונבנציות בקוד הוא eslint, הורידו ספרייה זו באמצעות npm, והשתמשו בה על מנת לשמור על קונבצניות (ועל מנת לא לעשות הכל באופן ידני).

## מבוא

עכשיו הגענו לחלק היותר נחמד של ההכשרה, כאן אנו נתעסק ממש עם אוטומציה.

נכיר את playwright, ספריית בדיקות יחסית חדשנית שפותחה מבית חברת מייקרוסופט, המאפשרת לנו לבצע בדיקות תצוגתיות על המערכת, כמו לחיצות על כפתורים, הזנת אותיות לשדות, ועוד המון דברים.

Playwright הוא כלי אומנם חדש, אבל החל לתפוס תאוצה מטורפת, עם קהילה שהולכת וגדלה כל יום.

כבר המון זמן הענף עובד עם הכלי המפורסם selenium, אולם, אחרי זמן מה הבנו של - selenium יש חסרונות כמו פונקציות חסרות, flakyיות, ועוד. על אף הקהילה הרחבה וההיסטוריה הרבה שלו, החלטנו לעבור לכלי שבאמת מבטיח איכות, מונע בעיות של חוסר יציבות בבדיקות, אינטואיטיבי ללמידה ופותר לנו המון בעיות בבדיקות.

לרשותכם שבועיים לעבור על כל ההכשרה ולסיים את כל התרגילים

בהצלחה 😊

## לפני שנתחיל

בחלק זה נלמד על פונקציונליות ב typescript, שנקראת async

פונקציונליות זו חשובה לנו, כיוון שאנו משתמשים בה כל הזמן בקוד שלנו.

אנו נרחיב על איך אנו משתמשים בה בהמשך, כרגע, גשו לתרגול

### חומר למידה

היכנסו לקישור הבא:

<https://www.youtube.com/watch?v=li7FzDHYZpc&ab_channel=RobertsDevTalk>

בסרטון זה, מוצג הסבר על פונקציונליות של promises, async, await

היכנסו לקישור הבא:

<https://www.atatus.com/blog/introduction-to-async-await-in-typescript/>

קראו את כל החומר הנלמד מהקישור

### תרגול

פתחו שוב את פרויקט ה – API שפתחנו כשניגשנו להכשרת Postman

על מנת לתרגל await ו async, נבצע כמה בקשות ב TS

קחו את כל הסעיפים שביצענו בהכשרת Postman, וכתבו אותם בפרויקט TS

בסיום, הגישו את הקוד בגיט, וקראו לחופף שלכם לעשות סקר קוד

#### הסעיפים שהיו ב Postman

1. בקשה ליצירת user חדש
2. בקשה ליצירת user קיים
3. בקשה לעדכון user קיים – ודאו ברשימה הגדולה
4. מחיקת user – וודאו שהוא לא קיים יותר ברשימה
5. הצגת מידע של user אחד מהרשימה

## הכשרה

### המערכת שעליה נכתוב בדיקות

אנו נבצע הכשרה מלאה על playwright ו cucumber, על פי המערכת הבאה:

<https://github.com/TestingTalks/react-app>

זהו פרויקט ממש נחמד, שיצר אדם נחמד בשם Cameron Bradley, האדם שפיתח הכשרה ב Udemy על השילוב של שלושת הכלים שאנו משתמשים בהם.

והבן אדם שבזכותו יש לנו תשתית אוטומציה עובדת.

#### הפעלת הפרויקט

פתחו git bash בשולחן העבודה שלכם, ובצעו clone לפרויקט.

לאחר מכן, כתבו את הפקודות הבאות:

yarn install

זה על מנת להוריד את כל הספריות של המערכת

yarn start

להריץ את המערכת – ניתן לכתוב פקודה זו ב cmd נפרד, כיוון שהיא חייבת לרוץ ברקע.

### התשתית

על מנת לא לעכב יותר מדי את ההכשרה, נביא לכם תשתית בסיסית מוכנה, עם הסבר כללי.

בצעו clone לקישור הבא למחשב שלכם:

<https://github.com/robin8360/Playwright-Training.git>

### הסבר קצר על התשתית

בואו נסתכל על עץ התיקיות והקבצים של התשתית

שימו לב שרוב הקבצים שנסביר כאן הם קבצי ברזל, משמע, אין צורך לגעת בהם או לשנות אותם, כי הם קבצי קונפיגורציה שאחראים על שלמותה של המערכת.

נשנה אותם רק במצבי קיצון, שזה אם הטכנולוגיות יתחדשו ויהיה צריך לשנות ספריות, ואיתם קינפוג מחדש, שהסיכוי שזה יקרה הוא אפסי, אבל לא בלתי אפשרי.

#### התיקיות

תיקיית ה config – בתיקייה זו, יש לנו את כל קבצי ה json, שמכילים בתוכם את כל הזיהויים של כל הדפים שבהם אנו עוסקים.

תיקיית ה src – בתיקייה זו, יש לנו את הקוד עצמו, את כל הבדיקות, וכל הפונקציות.

#### הקבצים

##### .babelrc

קובץ הגדרות לספרייה babel, כפי שהסברנו בהתחלה, ספרייה זו נועדה להמיר קוד ב JS חדש ל JS ישן.

להסבר יותר מפורט, מוזמנים לקרוא את הקישור הבא:

<https://babeljs.io/docs/presets#preset-options>

##### package.json

מכיל את כל הסקריפטים שמריצים לנו את הקוד, בנוסף לכל הספריות שיש לנו במערכת, כל פעם שנוריד ספרייה, היא תתווסף לרשימה.

##### run\_tests.bat

קובץ ההרצה שלנו, איתו אנו מריצים את הבדיקות.

כרגע הוא מאוד דל בתוכן שלו, מכיוון שהוא מריץ רק פקודה אחת מה package.json, אז אתם לא חייבים להשתמש בו כדי להריץ, אבל מוזמנים כדי להתרגל לשימוש בו כשהוא יתרחב.

##### tsconfig.json

קובץ הגדרות של typescript, כאן יהיו הגדרות כלליות כמו לאיזה תיקייה התוצרים של typescript הולכים.

##### playwright.config.ts

קובץ זה מכיל בתוכו הגדרות ל playwright, כמו האם לשים את המסך על מסך מלא, אילו דפדפנים אנו רוצים להריץ לבדיקות (chrome, firefox וכדומה...)

כמה זמן בדיקה מחזיקה עד שהיא נכשלת, ועוד...

יש הרבה מאוד דברים שאפשר להגדיר כאן והוא למעשה הקובץ המרכזי שלנו להגדרות של התשתית.

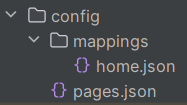
מוזמנים לקרוא עליו יותר בהרחבה בדוקומנטציה של playwright.

-------

בואו נפתח את שתי התיקיות המרכזיות שלנו.

#### מבט אל התיקיות

##### תיקיית config



תיקייה זו מחולקת ל 2 חלקים

**חלק ראשון** הוא תיקיית ה mapping, בתיקייה זו, יהיו את כל הקבצים של כל הזיהויים, כך שהם מחולקים על פי הדפים עצמם של המערכת.

אם נפתח את המערכת, היא תתחיל מדף הבית (home), כאשר הכתובת של דף הבית הוא רק /

אולם, אם נלחץ על כפתור ה create, הכתובת של הדף תהיה /tasks/create

###### עכשיו זה החלק שלכם להתנסות קצת עם התשתית

ולכן, נצטרך עוד קובץ לדף של הוספת איש קשר, עם כל הזיהויים אודות אותו הדף.

במערכת בסדר גודל יותר רחב, יש המון אלמנטים שחוזרים על עצמם בכמה מקומות

אם ניקח לדוגמא את הלוגו של המערכת כאן, נשים לב שהזיהוי שלה אותו הדבר גם בדף הבית וגם בדף יצירת איש הקשר, במצב כזה, ניצור קובץ, ונקרא לו

common.json

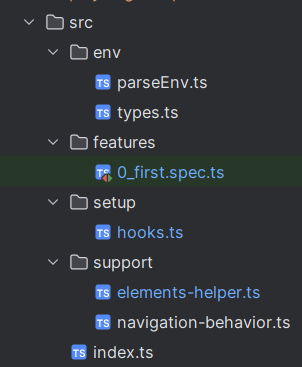
בקובץ זה, נשים את כל הזיהויים הכללים של המערכת, על מנת לא ליצור חזרתיות ולשים את אותו הזיהוי כל פעם בקבצי json שונים.

**החלק השני** הוא קובץ ה pages.json, הוא מכיל את כל הכתובות של כל הדפים, עם שם שתואם לאותו הדף.

למשל, אם עכשיו נוסיף את הדף של הוספת איש קשר לתיקיית ה mappings, נצטרך קודם להוסיף את הכתובת של הדף לקובץ ה pages, ולתת לו שם תואם, למשל create contact.

אחרי שהוספנו אותו לקובץ ה pages, אנחנו צריכים ליצור קובץ בתיקיית ה mappings, והשם של הקובץ חייב להיות תואם לשם שכתבנו בקובץ ה pages.

##### תיקיית src



תיקייה זו מחולקת למספר חלקים

###### תיקיית ה env

מכילה בתוכה קבצים שנועדו לקינפוג, יציבות ונראות של הקוד.

קובץ ה parseEnv מכיל פונקציות שמקבלות קלט של שם משתנה, ומחזירות את הערך שלו מקבצי הקונפיגורציה הקיימים (common.env, קבצי json).

קובץ ה types מכיל בתוכו את כל הטיפוסים היותר מורכבים של הקוד, וגם טיפוסים שחוזרים על עצמם, זאת על מנת בעיקר להציג את הקוד באופן יותר יפה ומובן לעין.

###### תיקיית ה features

תיקיית ה features מכילה בתוכה קבצים עם כל התכולות המרכזיות של המערכת.

למשל, הדף של הוספת איש קשר יכיל בתוכו מספר בדיקות (נקרא גם תרחישים):

1. פונקציה שמוסיפה איש קשר חדש
2. פונקציה שמזינה תווים לא חוקיים לתוך כל אחת מהשדות
3. פונקציה שמוודא את כפתור הביטול

לכרגע, קובץ spec כזה יכיל בתוכו בדיקות, שבתוכן יהיו את הצעדים שאנחנו עושים בבדיקה.

הצעדים שאנו עושים בבדיקה יהיו ה steps, אילו למעשה פונקציות קטנות, הכוונה בפונקציה קטנה היא פונקציה בעלת גג שורה אחת או שתיים, שמבצעת פעולה מאוד מינימלית, כמו לחיצה על אלמנט שהיא מקבלת, הזנת תווים לאלמנט וכדומה.

כרגע אנו לא משתמשים באופציה הזאת כי אפשר לכתוב הכל בבדיקה, אבל רצוי שאם יש לכם פונקציונליות שחוזרת על עצמה יותר מבמקום אחד, אתם מקבצים אותו לפונקציה וקוראים לה בבדיקה.

אתם יכולים ליצור אפילו קובץ שמטפל בצעדים האלו, או לשים אותו תחת element-helper אם הוא נראה לכם מתאים לקובץ הזה.

###### תיקיית ה setup

מכיל בתוכו קבצים שנועדו לאתחול, תפעול והרצת הבדיקות והמערכת.

קובץ ה hooks.ts מכיל בתוכו פונקציות לאתחול, הרצה וכיבוי של ה driver, וכניסה לאתר הראשי שלנו

ה driver הוא למעשה אובייקט, שנועד להפעלה של המערכת לבדיקות, הוא משתמש בchrome driver שאנחנו מביאים לו, ומריץ דפדפן שאיתו אנו יכולים לבצע פעולות, כמו למשל, להטעין עליו את הדף שאנחנו רוצים, לבצע פעולות כמו לחיצה על כפתור, הזנת טקסט, ועוד המון אפשרויות.

###### תיקיית ה utils

יכיל פונקציות עזר, על מנת לשפר את הנראות של הקוד (קוד ספגטי), ולכתוב פונקציות שיכולות לחזור על עצמם בהמון מקומות.

###### קובץ ה index.ts

אתחול של כל קבצי הקונפיגורציה/הזיהויים של האוטומציה, לתוך אובייקט הנקרא GlobalConfig.

מעולה, עכשיו, אחרי שעברנו על התשתית יותר בהרחבה, נצטרך ללמוד איך להתעסק איתה בפועל, וכאן, נצטרך להתחיל לעבוד עליה.

אז חגרו חגורות, כי עכשיו הגענו לחלק האמיתי.

## תרגול

לרשותכם מספר תרגילים לביצוע על התשתית המוכנה.

לא עברנו ממש בהרחבה על פונקציות של playwright ואיך להתעסק איתן, זאת כיוון שבחלק זה של ההכשרה, אתן לכם יד חופשית לחקור אודות פונקציות של playwright.

מוזמנים להסתכל בדוקומנטציה, על סרטונים, ועל כל דבר שעולה ברוחם.

אתן לכם כיוון על מה לחפש:

איך לקבל אלמנט בדף, איך ללחוץ על אלמנט, איך להזין טקסט בתוך אלמנט.

שימו ❤ לפני שאתם מתחילים, כתבו ב terminal בתיקייה של הפרויקט את הפקודות הבאות

npx playwright install

על מנת להוריד את כל ה driverים של playwright (chrome, firefox, webkit), עשוי לקחת כמה דקות להוריד

npm i

זאת על מנת להוריד את כל הספריות של הפרויקט לוקלית למחשב שלנו.

אחרי כל משימה שאתם מסיימים, גשו לחופף שלכם שיבצע לכם code review

### משימה ראשונה

בקובץ 0\_first.spec.ts

שנו את שם הקובץ שיקרא: 0\_homePage.spec.ts

כתבו בדיקה בקובץ זה, הבדיקה מקבלת page מטיפוס Page, שזה למעשה ה driver של chrome שעליו אתם מריצים את המערכת

הבדיקה תבצע בדף הבית את הפעולות הבאות:

1. הזנה בשורת החיפוש: Alea Nieves
2. וידוא שמוצג איש הקשר בעל הפרטים אודותיו
3. וידוא שמוצג רק איש קשר אחד ברשימה
4. לחיצה על מחיקת איש הקשר
5. ניקוי שורת החיפוש
6. וידוא שאיש הקשר לא מוצג ברשימה

שימו ❤ על מנת ליצור פונקציות עזר שיסדרו את הקוד, צרו תיקיית steps תחת src, ושם כתבו את פונקציות העזר שלכם.

### משימה שנייה

צרו קובץ spec חדש, שמו יהיה על פי התכולה שהולכת להיות במשימה הנ"ל.

הבדיקה הבאה הולכת ליצור איש קשר חדש, הזינו בפרטים כאוות נפשכם.

לאחר מכן, וודאו ברשימה שאיש הקשר נוצר.

### משימה שלישית

#### חלק ראשון

בתוך קובץ ה spec שיצרתם, צרו בדיקות שליליות,

בדיקות שליליות זה בדיקה רגילה, רק שהיא מוודא עבור דברים שהם לא חוקיים במערכת

כמו למשל, אם יש תווים שאסור להכניס לשדה, צריך שתהיה מוצגת הודעת שגיאה בהתאם.

במערכת הנוכחית, בכל השדות של יצירת איש קשר חדש, כאשר לא נזין כלום באותו שדה, תוצג הודעת השגיאה:

Error: The "(.\*)" field can't be empty.

בכל בדיקה שלילית על כל שדה, השאירו שדה אחד ריק, בזמן ששאר השדות יהיו עם תווים חוקיים.

#### חלק שני

כתבו בדיקה נוספת, בדיקה זו תזין תווים חוקיים לשדות (כמו יצירת איש קשר), אבל במקום לאשר את יצירת איש הקשר, לחצו על כפתור הביטול, ותוודאו שאיש הקשר לא נוסף לרשימה.

בדקו בשתי דרכים:

* כאשר הרשימה מוצגת כמו שהיא
* חפשו את איש הקשר וודאו שלא מוצג כלום ברשימה

### משימה רביעית

צרו קובץ spec חדש, על דף ה playground.

גשו לדף ה playground.

כאן, נרצה להתמקד בכמה אלמנטים, ולכן, משימה זו מחולקת למספר תרחישים קטנים, שכל תרחיש נוגע באלמנט אחר בדף.

התרחישים יהיו להלן:

#### תרחיש ראשון – Autocomplete Combo Box

1. הזינו שם תקין של סרט, ובחרו את האופציה הראשונה, וודאו ששם הסרט שנבחר מוצג בתיבת החיפוש.
2. מחקו את מה שכתוב בשורת החיפוש.
3. מבלי לחפש, בחרו את אחת מהאופציות, וודאו שהאופציה שבחרתם מוצגת בתיבת החיפוש.

#### תרחיש שני - Check Box

לפניכם 5 תיבות סימון, לחצו על כל אחת מהתיבות, וודאו את הדרך הכי אופטימלית לוודא שכל תיבה מסומנת.

אם אתם מתקשים, השתמשו ברמז לפניכם (בטלו את ההדגשה):

לכל תיבת סימון, יש אבא שהתגית שלו היא span, אם תרחיבו את הלשונית למטה שמציגה פרטים על כל אלמנט, תראו שכאשר אתם לוחצים על תיבת הסימון, יוצג ה CSS Value שנקרא color.

עליכם לגשת לתכונה זו של האלמנט, ולוודא שהיא בצבע הנכון כשהתיבה מסומנת.

על מנת להקל על החיפוש של הזיהוי של אותו ה span, גשו לקוד של המפתחים, והוסיפו לכל ה spans האלו data-id ייחודי.

#### תרחיש שלישי – Switch

לפניכם שני כפתורי מתג, אחד ניתן להזזה, והשני לא.

לחצו על האחד שניתן להזזה, וודאו שהוא פעיל.

עם האחד שלא ניתן להזזה, וודאו שהוא לא פעיל לגמרי – כלומר, שלא ניתן ללחוץ עליו.

#### תרחיש רביעי – Table

צרו רשימה ב typescript שמכילה את הרשימה המוצגת בדף, עם כל הפרמטרים.

עברו על כל הרשימה בדף, וודאו שהרשימה שמוצגת תואמת לרשימה שכתבתם.

דרך נוחה ליצירת הרשימה תהיה עם יצירת class.

כל הכבוד!

עברתם בהצלחה את הכשרת playwright 😁

אולם, יש לפנינו עוד כלי חדש

כלי זה הולך לשנות את נקודת המבט שלנו על כתיבת בדיקות אוטומציה.

אז קדימה, בואו נצלול לחלק הבא

# Cucumber

## לפני לפני שנתחיל – דף קונבנציות

אחרי שסיימנו את הכשרת playwright, ולמדנו איך לכתוב קוד כמו מקצועניים😎 נעבור לקונבנציות בהכשרה cucumber

פה יהיו בעיקר כללים לכתיבה נכונה בקבצי feature, וקבצי step-definitions

### קונבנציות להכשרה - קבצי feature

1. שמות הקבצים יהיו עם מספר בהתחלה, זאת על מנת לסדר את הקבצים באופן שבו נרצה להריץ, והשמות יהיו עם קו תחתון. דוגמא לשם קובץ:

0\_contacts\_actions.feature

1. מעל כל תכולה, נכתוב את האנוטציה regression – זאת על מנת שנוכל להריץ את כל התכולות בו זמנית (בעיקר בשביל בדיקות בלילה)
2. בין כל תרחיש חייב להיות רווח של שורה
3. אם התרחיש הוא Scenario Outline, אז ה Examples צריכים להיות עם tab אחד בין הכותרת לטבלה. הכותרת צריכה להיות מרווחת בינה לבין הסוף של התרחיש
4. תחת כל כותרת צריך להיות tab
5. ה background חייב להכיל את הצעד

Given i switch to the screen

על מנת לא למחזר את השורה, וכי זה הדרך שלנו להתחיל תרחיש (יש מקרים שבהם לא מתחילים בזה, אבל הם לא בהכשרה שלנו כרגע)

1. כאשר נרצה להריץ רק תרחיש אחד, נכתוב מעליו אנוטציה של dev
2. מילות המפתח של Cucumber באות בסדר הבא:
   1. Given – בתחילת התרחיש, מתחיל את הפעולה הראשונה לביצוע
   2. When – כאשר אנו רוצים לכתוב פעולה של המשתמש

When i click

* 1. And – ממשיך את מילת המפתח שמעליו, אפשר לשים כמה And אחד אחרי השני
  2. Then – כאשר אנו רוצים לוודא את הפעולות שעשינו

דוגמא לקובץ שעומד בקונבנציות

@regression

Feature: Contacts actions

  Background:

    Given i go to the page "home"

    And the element "home title" should contain the text "Contacts"

  @dev

  Scenario: Create a new contact

    When I click on the element "create new contact button"

    Then the element "create contact title" should contain the text "Create Contact"

### קונבנציות להכשרה – קבצי step-definitions

1. כל הקבצים יהיו תחת התיקייה step-definitions
2. התיקייה תהיה מחולקת לתתי תיקיות, כשלכל תיקייה יש קטגוריה משלה, לדוגמא:

צעד שלוחץ על כפתור יהיה תחת תיקיית actions

בעוד שצעד שמוודא טקסט יהיה תחת תיקיית assertions

1. צעדים יהיו גנריים ב 90 אחוז מהזמן, 10 אחוז זה אם יש צעד שבודק משהו ממש נקודתי
2. פונקציה של צעד תהיה מעט מאוד שורות קוד, כיוון שכל צעד אחראי על פעולה קטנה.

במידה ויש צעדים שהם יותר ארוכים, תהיה פונקציה שתקרא מתוך הצעד.

1. צעדים שמתקשרים לאותה קטגוריה, יהיו תחת אותו קובץ TS, דוגמא:

צעד שמוודא טקסט בשדה אחד, וצעד שמוודא טקסטים בכמה שדות, יהיו תחת קובץ שיקרא: validate-element-value.ts

1. בכל צעד, נעביר את המחלקה ScenarioWorld, שהוא אחראי על שמירה של ה driver, וכל המשתנים הגלובלים שלנו (GlobalConfig)
2. כאשר בצעד מועברים פרמטרים, הם צריכים להיות בדיוק באותו הסדר שאנו מקבלים מהשם של הצעד, והם צריכים להיות בעלי שמות משמעותיים, לדוגמא:

When(/^I click on the element "(.\*)"$/,

    async function (this: ScenarioWorld, elementKey: string) {

### כללי ניסוח לצעד

צעד צריך להתאים בשם שלו בדיוק למה שהפונקציה עושה, אם הצעד לא תואם, זה יכול ליצור סיבוכיות וחוסר הבנה.

הניסוח צריך להיות מובן, כך שנוכל לקרוא אותו משני הקבצים, feature ו TS

ניקח דוגמא לצעד שמובן בקובץ feature, אך לא מובן בקובץ TS

Then the "add contact button" is "enabled"

אולם בקובץ TS

Then(/^the "(.\*)" is "(.\*)"$/, async function (...)

זה מועד לפורענות, כיוון שאז אפשר לכתוב כל דבר שנרצה, כמו למשל

Then the "add contact button" is "visible"

פתרון לצעד הזה

Then the element "add contact button" has the enability "enabled"

אולם, הדרך היותר מומלצת לכתוב את הצעד הזה היא כך:

Then(/^the element "(.\*)" is "(enabled|disabled)"$/,

כלומר שיהיו לצעד רק שתי אפשרויות לבחור לצעד הספציפי הזה, ובקובץ feature הוא לא ימצא את הצעד אם לא כתבתם את המילים המתאימות

## מבוא

מתודולוגיית ה BDD – Behavioral Driven Development הינה תורת בדיקות, בה המערכת צריכה להיראות ולהתנהג לפי איך שהלקוח\המשתמש מצפה להתעסק איתה.

בפאן הבדיקות, זה בא לידי ביטוי בכך שאנו כותבים כל תרחיש כפי איך שהלקוח היה מצפה שהמערכת תתנהג, איך כל חלק צריך ואמור להתנהג, וזה בא לידי ביטוי בעיקר בעזרת cucumber.

כלי הניהול cucumber, נועד לכתיבת הבדיקות שלנו כפי איך שהמשתמש מצפה שהמערכת תתנהג, בוא אנו מגדירים צעדים ברורים בתוך כל תרחיש, ותרחישים ברורים לפי כל תכולה. כל הבדיקות יהיו כתובות באנגלית, כך שכל אדם, בין אם זה מפתח בצוות, הבוס שלנו, או מישהו רנדומלי מהרחוב (או מהבסיס, כי הם לא יכולים לבוא פיזית לבסיס מהרחוב...), יוכל להבין מה הבדיקות שלנו עושות.

מכיוון ש cucumber הוא כלי ניהול, הוא מחלק לנו את הבדיקות בצורה נוחה, יעילה, ונקייה. את הקוד כשלעצמו, משמע כל האוטומציה, אנו כותבים ב playwright.

אני מבינה שאני מתחילה לבלבל, ולכן, בוא ניגש לחלק המעניין 🤩

לרשותכם שבוע לעבור על כל ההכשרה ולסיים את כל המשימות

בהצלחה 😊

## הכשרה

### התשתית

כמו בהכשרת playwright, נביא לכם תשתית בסיסית של cucumber, שבה יהיו כל קבצי הקונפיגורציה של המערכת, וגם, יהיו כמה צעדים ובדיקות בשבילכם לחקור ולהריץ.

כמובן, שאם אתם רוצים להעמיק יותר על בניית התשתית המלאה, מוזמנים לרכוש את ההכשרה המטורפת שכתב Cameron Bradley, בקישור הבא:

<https://www.udemy.com/course/build-a-cucumber-playwright-typescript-automation-framework/>

עדיף לחכות למבצעים, כי ההכשרה כשלעצמה עולה המון כסף ☹️

קישור לתשתית:

<https://github.com/robin8360/Cucumber-Playwright-Training.git>

### הסבר קצר על התשתית

בעץ התיקיות וכל מה שטוב, נוספו לנו כאן כמה קבצים:

##### cucumber.js

קובץ הגדרות ל cucumber. כאן, אנחנו אומרים ל cucumber, שכאשר הוא מריץ את קבצי ה feature, הוא צריך לגשת למודל שאנו מייצאים לו, שזה תיקיית ה – dist, שבה יש את קבצי ה JS שהקוד צריך כדי לרוץ.

חסרון אחד שיש ל - cucumber.js, הוא שהקובץ הזה לא תומך בגרסת nodejs של ES6 או ב typescript כשפה, לכן, אנחנו משתמשים בספריות של babel, שנועדו להמיר קוד של JS חדש לגרסה ישנה.

##### .eslintrc.json

כפי שהראנו בהכשרת playwright, גם כאן יש לנו קובץ, שנועד לשמור לנו על הקונבנציות של קבצי ה TS. כל החוקים שכתובים כאן, כתובים גם [למעלה בקובץ](#_קונבנציות_להכשרה_–).

קראו את הקובץ על מנת להבין פחות או יותר את הסינטקס ומה ממה שיש שם רשום גם כאן בקובץ.

##### package.json

אולם הקובץ הזה היה כבר קיים גם בפרויקט הקודם, אבל כאן אני רוצה שנרחיב עליו קצת, יש כאן כמה דברים שאולי יראו מבלבלים וצריך לעשות כאן סדר מסוים.

תחת אובייקט ה scripts, נראה שהתווספה כמות לא קטנה של סקריפטים שאנו מריצים, נעבור עליהם אחד אחד על מנת להבין מה הן עושות:

**transpile –** פקודה זו מבצעת 3 פעולות:

* ניקוי של הקוד שלנו על ידי הרצת פקודת eslint
* מחיקה של תיקיית ה dist
* יצירה מחדש של תיקיית ה dist, ששם הקוד שלנו יהיה מאוחסן כקוד JS

למעשה אנחנו מבצעים כאן תנאים תחיליים להרצה של הבדיקות שלנו, כפי שאמרנו בתחילת ההכשרה, על מנת להריץ קוד של TS, אנחנו צריכים קודם כל להמיר את הקבצים לקבצי JS, ולאחר מכן, אם אנחנו עובדים עם גרסה חדשה של JS, צריך להמיר אותה ל JS ישן, באמצעות ספריית ה babel.

**precucumber –** פקודה זו מבצעת 3 פעולות:

* מוחקת את תיקיית ה reports
* מייצרת את תיקיית ה reports מחדש
* מייצרת קובץ reports.json ומכניסה לתוכו אובייקט ריק

כאן אנו מבצעים תנאים תחיליים בשביל תשתית ה cucumber, כאשר אנחנו רוצים ליצור קובץ, שישמש כקובץ הדו"ח בדיקות שלנו. כמו שב playwright יש דוח בדיקות שמציג אם יש בדיקות שנכשלו או עברו, כך גם ל cucumber.

**cucumber –** פקודה זו מריצה את הבדיקות שלנו, על ידי הרצת הפקודה cucumber-js. ובנוסף לכך היא מריצה את הפקודה transpile.

**postcucumber –** פקודה זו מייצרת את דו"ח הבדיקות, על ידי הרצת הקובץ cucumber-report.ts

**lint & lintandfix –** פקודות אילו בודקות clean code, ומתקנות בהתאם לפקודה הנבחרה

מעולה, עברנו על כל הסקריפטים של המערכת, אם יש משהו שאתם לא מבינים, מוזמנים לשאול את החופף שלכם או את CoPilot.

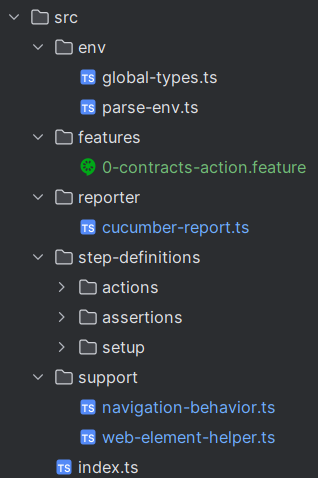
----

אם נפתח את התיקיות המרכזיות, נראה שהתווספו לנו כמה קבצים.

אל דאגה, נעבור עליהם אחד אחד.

#### תיקיית ה src

בתיקייה זו, נראה כמה שינויים מרעננים



זה נראה קצת מסובך, אבל ברגע שנתחיל לעבור תיקייה תיקייה, נצליח לסדר הכל כמו שצריך.

##### תיקיית ה reporter

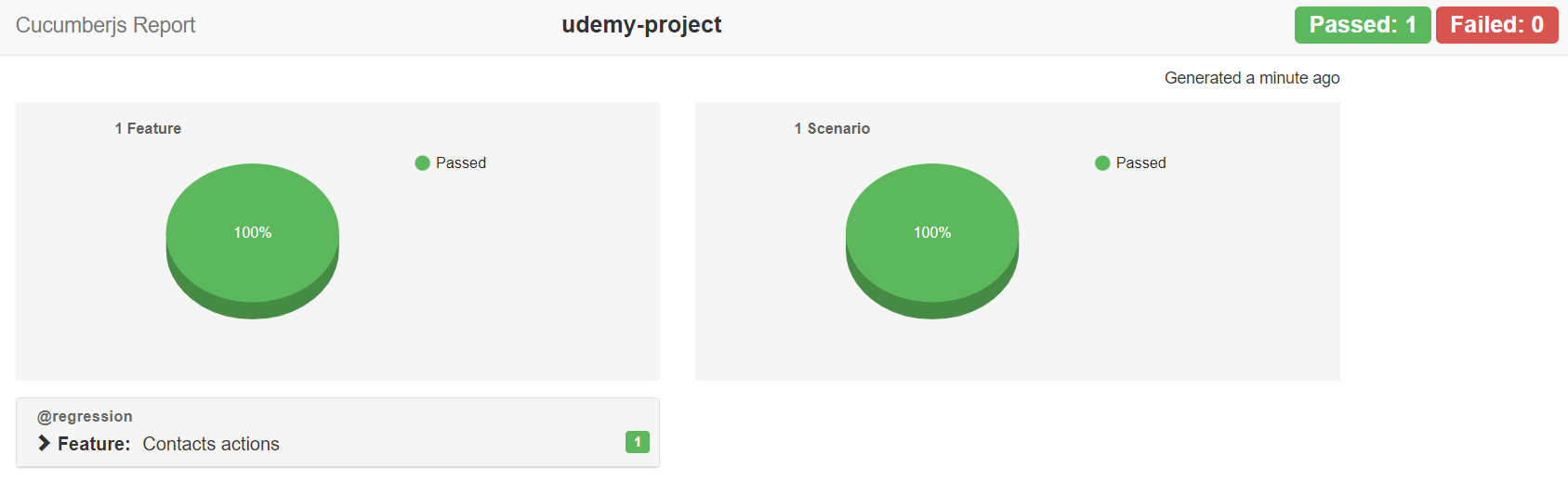
תיקייה שנוספה לנו היא תיקיית ה reporter

על אף שלכתוב בדיקות זה נחמד וכיף, בסוף אנחנו צריכים להציג את תוצאות הבדיקות לאנשים מעלינו, כיוון שהם לא מכירים את הקוד עצמו של הבדיקות.

כאן אנחנו מקנפגים את הפרמטרים של cucumber, שבסוף אחראים לייצוא של דוח הבדיקות שלנו.

למעשה, אחרי כל הרצה של הבדיקות שאנחנו מבצעים, נוצר לנו דוח תחת תיקייה בשם reports. הדוח הוא קובץ html, שמציג את התוצאות בצורה ויזואלית מאוד יפה.

דוגמא לדוח על הבדיקה מההכשרה:



##### תיקיית ה features

כמו שניתן לשים לב, תיקיית ה features שלנו השתנתה, פעם היו בה קבצי ts, ועכשיו, יש בה קבצי feature.

כל קובץ כזה, הוא תכולה כללית של המערכת, ניתן לחלק את המערכת לכמה תכולות, על פי מה שנראה הכי נכון.

את המערכת שאנו עובדים עליה אפשר לחלק לכמה חלקים:

1. דף הבית
2. דף יצירת איש קשר
3. גן המשחקים

כרגע חילקתי את זה לפי דפים, מכיוון שהמערכת יחסית קטנה ואין לה הרבה תכולות גדולות.

כל תכולה מחולקת לתרחישים שהיא מכילה בתוכה, כאשר תכולה זה רכיב במערכת, ותרחיש זה איך אני מתעסקת בתור משתמשת עם אותו רכיב, מה הפונקציות שאני יכולה לבצע אודות אותו רכיב.

אם ניכנס לקובץ שנמצא בתוך התיקייה, ה:

0\_home-page.feature

נראה כמה דברים חשובים בקובץ

@regression

Feature: Contacts actions

  Background:

    Given i go to the page "home"

    And the element "home title" should contain the text "Contacts"

  @dev

  Scenario: Create a new contact

    When I click on the element "create new contact button"

    Then the element "create contact title" should contain the text "Create Contact"

קודם כל, כפי שהצגנו בהתחלה, הייעוד של cucumber זה להפוך את הבדיקות לקריאות ומובנות על ידי כל מי שקורא, ולייעל את הקוד. לכן, השימוש בשפה האנגלית לכתיבת הבדיקות הופך את הכל ליותר מובן.

אנחנו נתחיל מכמה מילות מפתח חשובות של השפה:

Feature – המילה הראשונה בכל קובץ feature, היא למעשה אומרת שיש תכולה חדשה שאנחנו יוצרים, לכן, המילה הזאת מופיעה רק פעם אחת ובתחילת הקובץ.

השם של התכולה צריך להיות קצר ועניין, כמו contacts actions, שזה אומר, ביצוע פעולות סביב איש קשר.

מתחת לזה, אפשר לכתוב (לא חובה, מומלץ) תיאור קצר של מה הולך להיות בתכולה, איזה תרחישים כללים אנחנו נכתוב.

תכולה יכולה להכיל המון תרחישים, ומכאן, אנו מגיעים למילת מפתח הבאה שלנו.

Scenario – יצירת תרחיש חדש בתוך התכולה, ניתן ליצור המון תכולות תחת תרחיש אחד (כמובן, במגבלת הטעם הטוב).

תרחיש צריך להיקרות על פי מה שהוא הולך לעשות בכלליות, כמו לדוגמא, יצירת איש קשר חדש, לקחנו את התכולה, והוצאנו ממנה פונקציונליות שבאה לידי ביטוי בתרחיש.

כמובן, שתרחיש מציג בכלליות את הפונקציונליות, ומה שקורה בפועל, זה הצעדים שתחתיו, שהם מגדירים את מה שהוא עושה, ומכאן, נקפוץ אל מילת, אם לא מילות המפתח הבאות.

Given, And, When, Then, But – הצעדים של כל תרחיש, תרחיש מכיל את כמות הצעדים הנדרשת על מנת לכסות את הבדיקה שאנחנו צריכים לכסות במלואה, שהבדיקה היא מה שכתוב בכותרת התרחיש.

כל מה שאנחנו רואים תחת אותו תרחיש, אלו למעשה פונקציות, כל צעד כזה מקושר באמצעות cucumber לפונקציה בקוד, שהיא מבצעת את מה שהצעד אומר לעשות בפועל.

הצעדים יכולים להיות יותר מפורטים משאר הדברים שהצגנו, הם למעשה צריכים לתאר ממש אחד לאחד פעולות שמשתמש עושה בזמן שהוא בודק את המערכת.

בנוסף למילות המפתח שהסברנו כאן, נשים לב למשהו אחר בקובץ שלא הדגשנו, וזה הם האנוטציות.

יש לנו בקוד שתי אנוטציות עיקריות שבהם נשתמש במהלך כל הבדיקות שלנו, ובהכשרה:

regression – אנוטציה זו תהיה לפני מילת המפתח Feature, היא בעצם תהיה לפני כל התכולות שאנחנו ניצור, והיא נועדה להריץ את כל התרחישים מכל התכולות.

dev – אנוטציה זו תהיה לפני מילת המפתח Scenario, אנו נשים אותה על כל תרחיש אנחנו נרצה לבדוק, ובמידה והתרחיש כבר לא רלוונטי, נמחק משם אנוטציה זו.

זוכרים את הקובץ run-tests.bat?

בהכשרת playwright היינו כותבים רק את השם של הקובץ והוא היה מריץ את הבדיקה כמו שצריך.

עכשיו, בנוסף לקובץ הזה, אנחנו צריכים להעביר לו פרמטר, וזה בדיוק האנוטציות שדיברנו עליהם.

כאשר אנחנו בסביבת בדיקות, אנחנו נעביר לו את הפרמטר **dev**, על מנת לבדוק תרחישים ספציפיים שאותם נרצה לבדוק (אם אנחנו עכשיו עובדים עליהם, או אם היה באג ואנחנו רוצים לחזור אליהם, וכדומה). אולם, במידה ונרצה להריץ את כל הבדיקות, שזה נושא שנדבר עליו בהמשך, נעביר לו את הפרמטר **regression**.

יש עוד המון מילות מפתח ל cucumber שאנחנו משתמשים בהם בקוד שלנו, מוזמנים להרחיב עליהם בדוקומנטציה של cucumber, העתיקו את הקישור הבא:

<https://cucumber.io/docs/cucumber/>

מילות מפתח נוספות שצריך להכיר:

1. Rule
2. Background
3. Scenario Outline

**משימת כתיבה:**

כתבו כאן על כל אחת ממילות המפתח שברשימה, איפה כותבים אותן ומה הן עושות.

קראו לחופף שלכם לאחר שסיימתם.

##### תיקיית ה steps-definition

בואו נרחיב קצת את התיקייה הזאת

תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי

מעכשיו, התיקייה שמכילה בתוכה את הצעדים של המערכת תהיה מחולקת לפי קטגוריות:

1. צעדים שמוודאים
2. צעדים שמבצעים פעולות שונות
3. פונקציות שמאתחלות את ה driver
4. ועוד...

כל קובץ שנמצא בתיקייה זו, זה הקוד בפועל שאנחנו נכתוב, כלומר, מה שכל צעד מבצע בפועל.

פתחו את הקובץ click-element.ts

When(/^I click on the element "(.\*)"$/,

    async function (this: ScenarioWorld, elementKey: string) {

        const {screen: {page}, globalConfig} = this

        const element = getElement(page, elementKey, globalConfig)

        await element.click()

    }

)

ניתן לשים לב כאן לכמה פרמטרים

נתחיל ממילת המפתח שהייתה לנו בקובץ feature, שהיא ה When

לאחר מכן, בתוך מילת מפתח זו, יש לנו את השם של הצעד, בין שני התווים /^$/

בפועל, זה הקסם של cucumber, זה מה שמקשר את הצעד מהקוד, לצעד בקובץ feature, השם של הצעד חייב להיות אחד לאחד כמו שהוא כתוב בקובץ ה ts, כי אז, הוא לא מוצא את הצעד.

בתוך הפונקציה עצמה, אנחנו כותבים בדיוק את מה שהשם של הפונקציה אמור לעשות, שזה לחיצה על האלמנט שאנחנו מקבלים.

אז קודם כל, אנחנו מקבלים את האלמנט כאובייקט Locator, ולאחר מכן, אנחנו לוחצים עליו, כמו שעשינו בהכשרה של Playwright.

שימו ❤צעדים הן פונקציות שמבצעות פעולות ממש קטנות, לכן, צעד לא יעלה על יותר מ 5 שורות קוד. ואם כן, הוא או יתחלק לכמה צעדים, או, במידה והצעד צריך לעשות משהו מעבר לצעד רגיל, הוא יתחלק לפונקציות.

בסופו של דבר, אנחנו רוצים לעשות צעדים באופן הכי גנרי שאפשר, ולכן, אנחנו נשתדל לשים את הצעדים האלו בין כל הקבצים האלו.

כמובן, שבפרויקטים יותר גדולים, יהיו לנו צעדים שהם קשורים נקודתית לחלקים במערכת, נכון להכשרה זו, אנו נבצע צעדים גנריים לכל התרחישים שלנו.

###### תיקיית ה setup בתוך תיקיית ה step-definitions

בתיקייה זו, יש צעדים שונים מהצעדים שראינו עד כה.

אם ניגש לקובץ ה hooks, נראה שבמקום מילות המפתח של הצעדים הרגילים, יש לנו את המילים Before וגם After.

מילות מפתח אלו הם צעדים, שבאים לפני ואחרי כל תרחיש. בפונקציות אלו אנחנו נאתחל את ה driver, ונכבה אותו.

קובץ מאוד חשוב שיש לנו הוא ה world.ts

בקובץ זה יש לנו מחלקה שנקראת ScenarioWorld, שיורשת מהמחלקה World

הסבר קצר על הורשה: כאשר אנחנו מורישים ממחלקה, אנחנו בעצם יכולים להשתמש בכל הפרמטרים שיש לאותה המחלקה, במחלקה הנוכחית שלנו.

פונקציונליות שאנחנו משתמשים בה היא

setWorldConstructor(ScenarioWorld)

בואו נסתכל שוב על הצעד למעלה

When(/^I click on the element "(.\*)"$/,

    async function (this: ScenarioWorld, elementKey: string) {

        const {screen: {page}, globalConfig} = this

        const element = getElement(page, elementKey, globalConfig)

        await element.click()

    }

)

אם נסתכל על הפרמטרים שהפונקציה מקבלת, נראה שהיא מקבלת את השם של האלמנט, אבל בנוסף לזה, היא מקבלת את המחלקה ScenarioWorld.

זה קורה בעזרת הפונקציה של setWorldConstructor, היא מגדירה את המחלקה הזאת לכל הצעדים של cucumber, ולכן, ניתן בעזרתה לגשת לפרמטרים שיש לנו תחת המחלקה, שזה ה driver, וה globalConfig.

הסברנו בהכשרת playwright על ה driver, אולם, יש לנו פרמטר נוסף במחלקה, שנקרא globalConfig.

globalConfig הינו פרמטר שמכיל לנו את כל הזיהויים במערכת, ואת כל הדפים שיש במערכת.

הוא למעשה מכיל את כל קבצי ה json שיש לנו תחת התיקייה config.

זאת כדי שבסוף, כשאנחנו מעבירים בצעד את השם של המשתנה, הוא יעבור על כל קבצי ה json שיש ב globalConfig, וייקח את הזיהוי על פי השם שנתנו לו.

את האתחול של globalConfig, אנו מבצעים בקובץ index.ts

##### קובץ ה index.ts

אם נסתכל על הקובץ, נראה שנוסף לנו בסוף הקובץ עוד כמה שורות של קוד.

const common =

    `./src/features/\*\*/\*.feature \

    --require-module ts-node/register \

    --require ./src/step-definitions/\*\*/\*\*/\*.ts \

    --world-parameters ${JSON.stringify(worldParameters)} \

    -f json:./reports/report.json \

    --format progress-bar`

בחלק זה, אנחנו מגדירים את כל מה שצריך להרצה של ה cucumber, שזה:

1. קבצי ה feature
2. קומפיילר של typescript להרצה של קבצי ה step-definitions
3. קבצי ה step-definitions, כדי שקבצי ה feature ידעו לאן לגשת בזמן ריצה בקוד
4. את ה reports שבסוף מייצאים דוח
5. את ה globalConfig, כדי שיהיה לו את כל הפרמטרים החשובים, שזה הדפים והזיהויים של המערכת, בזמן ריצה של הבדיקות.

בסוף, אחרי שאנחנו לוקחים את כל הקונפיגורציה הזאת, אנחנו שמים אותה עם האנוטציות שאיתם אנחנו מריצים את הבדיקות.

const dev = `${common} --tags @dev`

const regression = `${common} --tags @regression`

export {dev, regression}

ובסוף, אנחנו מריצים אנוטציות אלו בקובץ run-tests.bat.

עברנו על כל התשתית הבסיסית של האוטומציה שלנו, כל הכבוד🤩

עכשיו, נעבור להכשרה בפועל.

## תרגול

המשימות בחלק זה יהיו דומות ל playwright, אבל עם יותר חשיבה פתוחה.

על כל משימה שאתם מסיימים, תקראו לממונה עליכם ל code review.

### משימה ראשונה

גשו לקובץ 0-contracts-action.feature

המשיכו את התרחיש של יצירת איש קשר.

לאחר מכן, הוסיפו עוד 2 תרחישים:

1. מחיקה של איש קשר לבחירתכם, ווידוא שהוא לא קיים ברשימה
2. עריכת איש קשר קיים, וידוא שהוא השתנה ברשימה עם כל הפרמטרים שלו – שימו לב שאתם צריכים לוודא את הפרמטרים הראשוניים שלו, כשאתם פותחים את המשתמש הקיים לעריכה

חשבו על עוד תרחיש אחד שאפשר לעשות על התכולה של איש קשר, וכתבו אותו.

בנוסף, כתבו בדיקות שליליות לעריכת איש קשר – שדות ריקים.

### משימה שנייה

גשו למשימה הרביעית בהכשרת playwright, ובצעו אותה על התשתית הנוכחית.

קישור למשימה: [משימה רביעית](#_heading=h.3cqmetx)

### משימה שלישית

בדף ה playground, יש לנו כמה אלמנטים שנועדו להעביר אותנו בין דפים.

צרו 3 תרחישים, כאשר כל תרחיש יעבוד על אלמנט אחר בדף

#### תרחיש ראשון – Open Tab

לחצו על הלינק, וגשו אל הלשונית החדשה שנפתחה.

בתוך הלשונית, לחצו על כפתור ה create, וודאו שנפתח הדף של יצירת איש קשר.

#### תרחיש שני – Open Window

לחצו על הלינק, גשו לדף החיצוני החדש שנפתח.

בתוך בדף החדש, לחצו על כפתור ה create, וודאו שנפתח הדף של יצירת איש קשר.

#### תרחיש שלישי – IFrame

גשו לפריים של דף הבית דרך דף ה playground.

בתוך הפריים, לחצו על כפתור ה create, וודאו שנפתח הדף של יצירת איש קשר.

כל הכבוד! 😁

סיימתם את כל הכשרת האוטומציה, אתם יכולים להתגאות בעצמכם!

גשו לחופף שלכם להמשך הכשרה על המערכת עצמה.