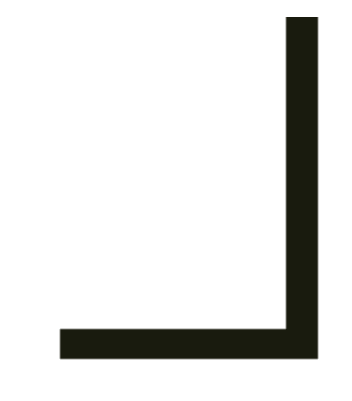


**Webpack**

**Workshop**



**לפני שמתחילים**

לפני שנתחיל ללמוד ולחקור על הדבר המופלא הזה שנקרא: "Webpack", בואו נחליט ביחד על המטרה שלנו בסדנה הזאת, או יותר נכון מה לא המטרה של הסדנה הזאת:

**הסדנה לא נועדה להפוך אותנו למומחי-על בנושא.** רגע אז למה אנחנו עוברים אותה?

* הסדנה נועדה להביא את מה שצריך תוכניתן הWeb הממוצע לדעת בנושא (וקצת מעבר 😊).
* בסדנה נלמד איך להשתמש בדוקומנטציות באתרים כמו Github וללמוד מהן!
* בסדנה נרחיב את הידע שלנו בנוגע לפיתוח בסביבת Dev!

מבנה הסדנה מורכב מכמה שלבים:

1. נלמד Webpack בדרך הידנית, על מנת להבין לעומק מה קורה מאחורי הקלעים.
2. נלמד Webpack בדרך האוטומטית!

הסדנה עצמה מתחלקת ל3 חוברות: חוברת למתחילים, חוברת למתקדמים וחוברת למתקדמים מאוד– אתם כרגע בחוברת ל**מתחילים.** לאחר שתסיימו את החוברת הזו – תעברו לחוברת למתקדמים.

המטרה של החוברת למתחילים היא להכיר לכם את עולם הWebpack, לטעום ממנו קצת ולהבין איזו יכולות עצומות יש בו.

לפני שאתם מתחילים וודאו שאתם למדתם את הנושאים הבאים:

1. HTML/CSS/JS.
2. NPM.
3. בקשות API (HTTP).

אז עכשיו שהכל סגור – בואו נצא לדרך!

**מה זה Webpack?**

בואו נחשוב על הסיטואציה הבאה:   
אנחנו כותבים אפליקציה web-ית, אנחנו רוצים להיעזר בשתי ספריות JS חיצוניות. נשמע פשוט לא?

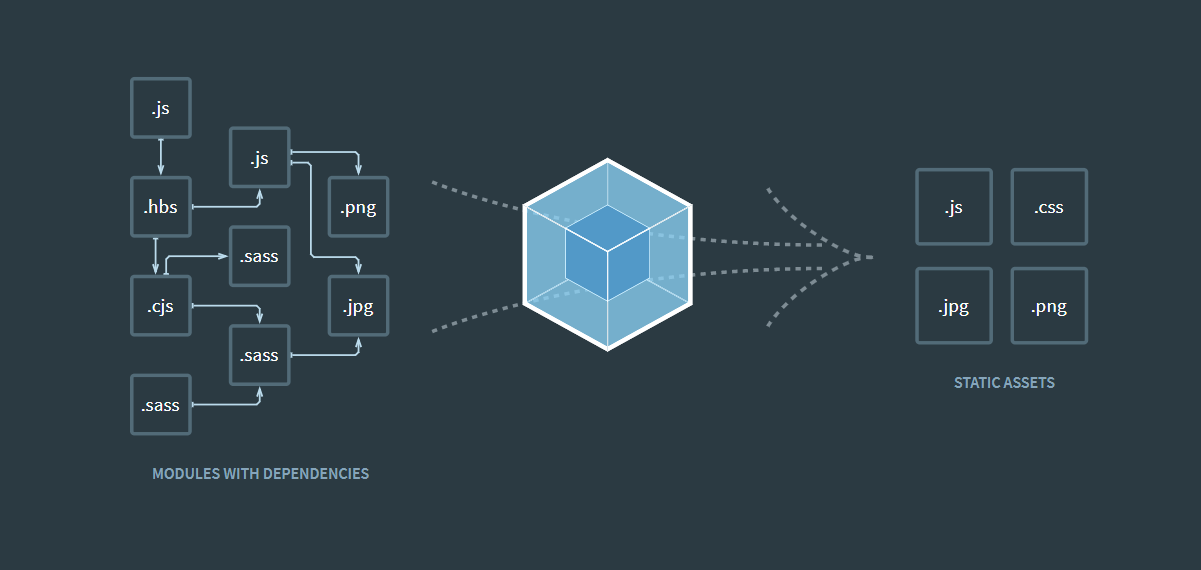
מה היינו עושים פעם (הדרך הישנה)?

היינו הולכים לקובץ הhtml שלנו ושמים שם תגית script עם הקישור לקובץ JS של הספרייה החיצונית וזה עבד מעולה, אבל הבעיות המרכזיות בשיטה הזאת הן:

* אם היה מספר גדול יותר של ספריות – מספר גדול יותר של קבצי JS שצריך לקשר לקובץ ה html, במקום 2 ספריות היה לנו 1000. אפילו 100 – לעקוב אחריהן ולזכור מה תלוי במה ומה לא יעבוד ללא מה, זה כמעט בלתי אפשרי!
* כל הסיפור הזה נוגד את כל העיקרון שב Rich Client – העיקר הוא הJS והכל עובר דרכה ולא דרך הhtml!

אז השיטה הזאת היא טובה, אבל יכולה להיות **טובה יותר.**

אז מה הפתרון שWebpack מציע?

Webpack זה ספריית NodeJS שמייצאת קבצים בודדים המאגדים מודול שלם אשר מבוסס על גרף של תלויות. במילים פשוטות – המטרה העליונה של Webpack זה לקחת הרבה מאוד קבצים עם הרבה מאוד תלויות ולייצא קבצים בודדים מהם, או כמו שהאתר הרשמי של החברה מעיד:

הספרייה מאוד פופולארית בקרב מפתחי הWeb כדי לייצא קבצים מאוגדים בדיוק בגלל הבעיות שהעלנו מקודם. כמו כן – חשוב לציין כי המוצר של Webpack הוא לא יחיד במינו וישנן עוד ספריות אשר יודעות לעשות בדיוק את זה לדוגמא: rollup.

**הבנת מבנה הפרויקט הידני**

עכשיו נעבור צעד צעד בתהליך פתיחת פרויקט חדש שכולל Webpack, שימו לב שעליכם לבצע זאת:

1. **אתחול פרויקט –** נפתח תיקייה חדשה ובה נכנס לCMD ונריץ את הפקודה הבאה:

npm init

מה שהפקודה הזאת עושה זה ליצור קובץ Package.json חדש. קובץ זה מכיל את כל התלויות של הפרויקט שלנו ובנוסף גם מכיל מידע אודות הפרויקט עצמו כמו: שם הפרויקט, כותבי הפרויקט, גרסה, רישיון וכדומה.   
במידה ותרצה להרחיב את הידע בנוגע לקובץ זה, אנחנו ממליצים לקרוא [פה](https://medium.com/@Nasita_Haque/package-json-cheat-sheet-4fe1b8baa102) [ופה](https://areknawo.com/whats-what-package-json-cheatsheet/).

לאחר הרצת הפקודה - יופיעו לכם כמה שאלות על הפרויקט – הכניסו את הפרטים שלכם ובמידה ואתם לא רוצים אתם יכולים פשוט ללחוץ Enter וזה ישים ערך דיפולטי.

לאחר מכן נתקין את הספריות **Webpack** ו **Webpack-Cli** בעזרתם נעבוד, נעשה זאת באמצעות

הפקודות, כמו כן ניתן להתקין אותן גם באופן גלובלי – חפשו על כך באינטרנט:

npm install webpack --save-dev

npm install webpack-cli --save-dev

עכשיו נכנס לקובץ Package.json ונראה משהו בסגנון כזה:

**{**

"name"**:** "project-templete"**,**

"version"**:** "1.0.0"**,**

"description"**:** ""**,**

"main"**:** "index.js"**,**

"scripts"**:** **{**

"test"**:** "echo \"Error: no test specified\" && exit 1"

**},**

"author"**:** ""**,**

"license"**:** "ISC"**,**

"devDependencies"**:** **{**

"webpack"**:** "^4.30.0"**,**

"webpack-cli"**:** "^3.3.2"

**}**

**}**

אנחנו יכולים לראות שיש לנו מאפיין שנקרא "Scripts" והוא מכיל אובייקט Json, שמורכב מ:

"test"**:** "echo \"Error: no test specified\" && exit 1"

בעצם המחרוזת הראשונה, ה"test" זה שם הסקריפט והשנייה זה הסקריפט עצמו, אותו נפעיל באמצעות הפקודה:

npm run שם סקריפט

מה שקורה פה בעצם זה שאנחנו אומרים למחשב להריץ (To Run) סקריפט מסוים.

**לשם התרגולים שלנו בסדנה הזאת –** נשנה את הסקריפט של הtest שערכו יהיה "Webpack" כך:

"test"**:** "webpack"

הפקודה Webpack בעצם מפעילה את הספרייה ומתחילה ליצור את הקבצים, לכן **במהלך התרגול** כדי לוודא שמה שכתבנו באמת עובד נריץ את הפקודה [אם תריצו עכשיו תיזרק שגיאה]:

npm run test

אז בואו נבדוק אם הSetup של הפרויקט הידני שלנו באמת עובד ונתחיל בתרגיל הראשון אשר נמצא בעמוד הבא.

**שימו לב!** בתוך כל תיקייה של תרגיל יש תיקיית EXRC. פתרו את התרגיל בתיקייה זו ולא בתיקייה חדשה שלכם.

**תרגיל #0 – "הצגת יכולות"**

1. לאחר "שהכנו את השטח", כלומר ביצענו את כל הצעדים המפורטים למעלה - **ניגש לתיקייה של הפרויקט החדש שלנו**.
2. **ניצור קובץ חדש בשם "webpack.config.js"**, כאן יהיו כל הקונפיגורציות של תהליך איגוד ואיחוד הקבצים שנעשה על ידי ספריית הWebpack.
3. **בקובץ נכתוב את הדבר הבא** (ללא ההערות עם הספרות):

**const** path **=** require**(**"path"**); // I**

module**.**exports **=** **{**

entry**:** "./title.js"**, // II**

output**:** **{**

filename**:** "main.js"**, // III**

path**:** path**.**resolve**(**\_\_dirname**,** "dist"**) // IV**

**}**

**};**

עכשיו בואו נסביר מה יש לנו כאן –

1. בשורה הזאת – אנחנו מייבאים את המודול "path", שבסה"כ עוזר לנו לעבוד עם קבצים, תיקיות ועצי תיקיות, עליכם לקרוא [כאן](https://stackoverflow.com/questions/9901082/what-is-this-javascript-require) -כדי להבין את הסינטקס של require.
2. בשורה הזאת – אנחנו קובעים את הקובץ הראשון שלנו (מסוג js) שהוא נקודת ההתחלה של גרף התלויות של הפרויקט שלנו, כלומר – הוא הקוד הראשון שבו אנחנו קוראים לקוד אחר וספריות אחרות וממנו נוצר הסיבוכיות והתלותיות בין הספריות. הוא הקובץ הראשי של התוכנית – אם נקביל את זה למה שלמדנו בVue – הEntry, הנקודת התחלה שלנו הוא הקובץ main.js שבו נמצאת הקומפוננטה הראשית שלנו – שימו לב שבתרגיל זה אתם מקבלים את הקובץ title.js ממומש.
3. שם הקובץ אליו יתאגדו כל התלויות, כלומר – בסוף התהליך כל התלויות והספריות יאוגדו לקובץ אחד תחת שם זה.
4. הנתיב בו הקובץ של הסעיף הקודם ייווצר.
5. **ניצור קובץ בשם index.html ובו ניצור תגית script שמקשרת את הקובץ JS שנוצר לנו אחרי תהליך האיגוד**, לפי ההגדרות שלנו בקובץ webpack.config.js בסעיפים III ו IV.
6. **נכנס לcmd ונריץ את הפקודת בדיקה שלנו** (מי שלא זוכר – שיבדוק עמוד למעלה😊).
7. **נכנס לקובץ index.html ואם הכל עבד- תקפוץ לנו הודעה**!

שימו לב - אתם מקבלים את קבצי הHTML וה JS מוכנים – עליכם רק לדאוג לקונפיגורציה.

**תרגיל #1 – "מטפלים בקבצים סטטיים":**

הכל טוב ויפה עד עכשיו, הראנו שאנחנו יכולים לאגד קבצים JS ביחד ולייצא הכל ביחד, אבל מה עם שאר סוגי הקבצים? נניח תמונה? אנחנו צריכים לחשוב איך אנחנו עושים את זה – אולי בJS קלאסי זה פשוט? נעשה משהו כזה (בקובץ title בו נוכל ליצור את התלות בתמונה):

// Require the image

**const** **image** **=** require**(**"./assets/image.png"**);**

// Getting body element and create new image element

**const** bodyElement **=** **document.**querySelector**(**"body"**);**

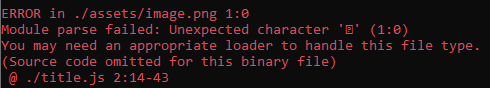
**const** newImageElement **=** **document.**createElement**(**"img"**);**

// Set the new image element to the image we've required

newImageElement**.**src **=** **image;**

// Add the new image element as a child to the body element

bodyElement**.**appendChild**(**newImageElement**);**

בסה"כ ייבאנו את התמונה, יצרנו תגית והוספנו אותה לbody – פשוט לא? אפילו אפשר להשתמש בקובץ קונפיגורציות שיצרנו בתרגיל הקודם! יאללה בואו נבדוק...

אז קפצה לנו שגיאה שאומרת שספריית הWebpack לא יכולה לטפל בקובץ מסוג זה, וש- "אנחנו צריכים Loader מתאים כדי לטפל בסוג קובץ זה".

אבל מה זה Loader?

Webpack לבדו יודע להתמודד עם JS בלבד – אז כשאנחנו רוצים לאגד קבצים מסוגים שונים כמו תמונות, CSS וכדומה הוא צריך "כלי" חיצוני שיעזור לו. אז Loaderים הם הפתרון לכך – Loader הוא כלי הנכתב בNodeJS ומיועד לספריית Webpack ועוזר לו לאגד או לקמפל סוג מסוים של קבצים ומידע. [כאן](https://github.com/webpack-contrib/awesome-webpack#loaders) תוכלו לקרוא על המון Loaderים – בחלקם גם נשתמש בסדנה!  
כדי לקבל את הפקודה בעזרתה נתקין את אותם Loaderים – נחפש את שמו בGoogle, כנראה

אז אנחנו יודעים מה הפתרון שלנו, איך משתמשים בו? –

1. נלך לקובץ הWebpack.config.js ולאחר האובייקט output נשים פסיק ו**נוסיף את קטע הקוד הבא:**

module**:** **{**

rules**:** **[{ // I**

test**:** **//, // II**

use **:** **[**“”**] // III**

**}]**

**}**

אז בואו נתחיל לפרש מה יש לנו כאן:

1. שורת הקוד הזאת היה ההתחלה של פסוקית בשם Rules, אשר מכילה את החוקים אליהם יהיה כפוף תהליך האיגוד כמו עבור אילו קבצים יעברו דרך איזה Loader, סדר הLoaderים ואפילו להגדיר נתיב עבור כל תהליך העובר דרך כל Loader.
2. בשורה זאת – אנחנו מגדירים אובייקט test שמכיל ביטוי רגולרי והוא מגדיר שכל קובץ (שהוא חלק מהתלויות של הפרויקט שלנו) העונה על ביטוי רגולרי זה יעבור את התהליך המפורט בפסוקית ה use (עליה נדבר בסעיף הבא). [כאן](https://www.rexegg.com/regex-quickstart.html) תוכלו להיזכר בחוקים של ביטויים רגולריים.
3. בפסוקית זו אנחנו מגדירים דרך איזה Loaderים יעברו הקבצים העונים על הביטוי הרגולרי (בביטוי הקודם).
4. עכשיו, **נכתוב את הביטוי הרגולרי המתאים** (בפסוקית II) בין שני ה / , כדי שהקובץ שבתיקיית assets יענה עליו. איך ניצור ביטוי כללי? רמז – סוג הקובץ הוא png.
5. נכניס בין המרכאות שבפסוקית III את שם הLoader שלנו – "file-loader", זכרו להתקין אותו!
6. נריץ את הפקודת בדיקה שלנו ונכנס לקובץ הhtml שלנו לבדוק אם זה עובד!
7. מה אנחנו רואים כאן? תהליך האיגוד עבר בהצלחה, הקוד שלנו בקובץ title.js עבר בהצלחה – עובדה שנוספה תגית img לקובץ, אבל אנחנו עדיין לא רואים תמונה. אם נכנס לDevTools של Google Chrome נראה כי קפצה לנו שגיאה שאומרת שהקובץ לא נמצא! אז מה יכולה להיות הבעיה? - אם נכנס לתיקיית dist שנוצרה לנו לאחר תהליך איגוד הקבצים (שם יהיו כל הקבצים המאוגדים) ואנחנו רואים את התמונה שלנו שם, אבל אם אתם זוכרים בקובץ title.js בו הוספנו אלמנט של תמונה לקובץ הhtml שלנו – אנחנו פונים בשורה הראשונה לתמונה כשהיא נמצאת תחת תיקיית assets. למה זאת בעיה – מאחר וכאשר אנחנו מקשרים את קובץ הJS המאוגד – הוא מחפש את התמונה באותו נתיב, אשר לא קיים בתיקיית dist – כלומר הוא מחפש את התמונה בתיקייה שלא קיימת (dist/assets) בעוד התמונה נמצאת בנתיב אחר. מה אפשר לעשות? – **נשנה קצת את פסוקית הuse של הLoader שלנו שתראה כך:**

use **:** **[{**

loader**:** ""**, // I**

options**:** **{ // II**

outputPath**:** ""**, // III**

publicPath**:** "" **// IV**

**}**

**}]**

טוב אז הפסוקית שלנו קצת השתנתה, בואו נראה מה יש לנו עכשיו:

1. אנחנו מכירים כבר את השורה הזאת – בסה"כ לציין באיזה Loader מדובר.
2. תחילת פסוקית options – כאן הדברים החדשים מתחילים, זאת הפסוקית שמאפשרת התאמה אישית של חלק מתהליכי הLoader כמו שינוי הנתיב אליו יועברו הקבצים שעברו דרך הLoader.
3. פסוקית זאת קובעת לLoader ליצור תיקייה ספציפית לסוג הקבצים שעוברים דרך הLoader הזה – במקרה שלנו אנחנו צריכים ששם התיקייה החדשה יהיה "assets".
4. פסוקית זאת נועדה לכוון את הדפדפן לתיקייה שהגדרנו מקודם בעת שהוא מחפש את הקבצים, כלומר – להגדיר לדפדפן את הנתיב הנכון (במקרה שלנו "dist/assets").
5. **עכשיו נריץ שוב את פקודת הבדיקה –** אם עשינו הכל כשורה, נפתח את קובץ הhtml שלנו ונראה את התמונה.

שימו לב - אתם מקבלים את קבצי הHTML וה JS מוכנים – עליכם רק לדאוג לקונפיגורציה.

**תרגיל #2 – "מטפלים בקבצי CSS":**

כפי שאנחנו כבר יודעים אפליקציה Webית היא לא רק html וJS – צריך להוסיף קצת CSS כדי להפוך את הכל ליפה יותר. אז בתרגיל הזה נכיר עוד כמה Loaderים חדשים!

1. **נתקין את שני הLoaderים החדשים שלנו: "style-loader", "css-loader", חפשו עליהם באינטרנט!**
2. בקובץ Webpack.config.js **נוסיף למערך ה rules שלנו את הLoaderים הנ"ל**, כפי שלמדנו איך להוסיף בתרגיל שעבר.
3. **כתבו את הביטוי הרגולרי** ש "יתפוס" לנו את כל קבצי הcss.
4. **נריץ את פקודת הבדיקה שלנו ובמידה והתהליך עבד נראה בקובץ הhtml שלנו אנימציה מגניבה.**

שימו לב - אתם מקבלים את קבצי הCSS, HTML וה JS מוכנים – עליכם רק לדאוג לקונפיגורציה.

**תרגיל #3 – "משנים נקודות כניסה":**

בדרך כלל למערכת שלנו יהיה יותר קוד מאשר התרגילים הקטנים פה, לכן במידה ואנחנו רוצים לפצל את הקבצים המאוגדים מסיבות כמו: הגבלת גודל הקובץ המאוגד, חלוקה לוגית– שלכל גרף תלויות יהיה קובץ מאוגד משלו וכדומה – יש לנו אפשרות להוסיף עוד "נקודות כניסה" – כלומר עוד נקודות להתחיל מהן את תהליך האיגוד. אבל איך נעשה את זה?  
אז עכשיו – אנחנו נכריח את הWebpack שלנו ליצור עוד Entry Point.

1. בתרגול הזה יש לנו 2 קבצים: "alert\_screen.js" ו"webpack\_animation.js", שניהם דואגים לדברים שונים: אחד לאנימציה שהייתה בתרגיל הקודם, כלומר גרף התלויות שלו הוא בCSS והשני להתראה למסך שהייתה בתרגיל #0. אז הצעד הראשון שלנו הוא **לקנפג את נקודות הכניסה החדשות שלנו, לכן נשנה את זה בקובץ הwebpack.config.js:**

entry**:** **{**

first\_entry\_point\_name **:** ""**, // I**

another\_entry\_point\_name**:** ""

**},**

output**:** **{**

filename**:** "[name].js"**, // II**

path**:** path**.**resolve**(**\_\_dirname**,** "dist"**)**

**}**

אז מה יש לנו כאן?

1. אנחנו יכולים לראות שהשינוי המרכזי ממה שהיה לנו עד עכשיו זה שבמקום פשוט מחרוזת פסוקית Entry מקבלת אובייקט במבנה של:

**שם נקודת ההתחלה** : **"נתיב לקובץ שממנו נתחיל" –** כך אנחנו בוחרים להוסיף עוד Entry לתהליך האיגוד שלנו.

1. השינוי השני שלנו הוא שבשם הקובץ אליו נייצא את התלויות המאוגדות, במקום "main" – רשום עכשיו "[name]", מה שיקרה זה שבאופן אוטומטי, הקבצים ייווצרו תחת השם שנתנו לנקודת ההתחלה בסעיף הקודם.

מה שעליכם לעשות זה **להעתיק את הקטע קוד החדש לקובץ קונפיגורציות שלכם, לתת שמות ונתיבים מתאימים** לפסוקית הEntry (I) **ולשנות את פסוקית הFilename** למה שלמדנו הרגע.

1. **ליצור קובץ HTML משלכם ובו תגיות הסקריפט המקושרות לקבצי הJS** שייווצרו בתהליך האיגוד שלנו.
2. **נריץ את פקודת הבדיקה שלנו** ואם קופצת לנו הודעה ואנחנו גם רואים את האנימציה – הצלחנו!

**תרגיל #4 – "די עם קבצי HTML"**

אז ראינו שאנחנו יכולים לחלק לקבצים שונים את גרפי התלויות שלנו, דבר שיעזור לנו כשנרצה להגביל את גודל הקובץ או לחלק את גרף התלויות לכמה גרפים קטנים או חלקים לוגיים קטנים, אבל דבר זה גורר בעיה אחת מרכזית – על כל קובץ שאני אוסיף אני אצטרך ליצור תגית script ולכתוב שם את הנתיב לקובץ המאוגד (סעיף 2 בתרגיל הקודם), אז לא די שזה יכול לגרור שגיאות בריצה אם טעינו בנתיב, זה גם מעיק להוסיף תגית כל פעם!

מה אפשר לעשות? - אז בתרגיל הזה נכיר את הפתרון שלנו: "Plugins" – כאשר Loaderים לא מספיקים לנו כדי לשנות או להוסיף יכולות לWebpack שלנו, נוכל להוסיף תוספים שירחיבו לנו את סט הפונקציונליות הבסיסיות שספריית הWebpack מספקת – בתרגיל זה נכיר תוסף כזה.

איך נוסיף תוסף?

1. תחילה **נוריד את התוסף** שלנו.
2. נלך לקובץ הWebpack.config.js **ונוסיף את קטע הקוד** הבא:

**const** Plugin **=** require**(**"Plugin-name"**); // I**

.

.

.

.

plugins**:** **[ // II**

**new** Plugin**({ // III**

**// IV**

**})**

**]**

אז מה יש לנו כאן?

1. בתחילת קובץ הקונפיגורציות נוסיף את פקודת הRequire, שמייבאת לנו את התוסף, שימו לב להתקין את התוסף עם הפקודה המתאימה ולתת שם משמעותי למשתנה אליה אנחנו מייבאים אותו.
2. לאחר פסוקית הmodule בקובץ שלנו נוסיף מערך חדש בשם plugins בו ניצור מופעים חדשים של התוספים שלנו, כפי שנראה בסעיף הבא.
3. יצירת מופע חדש של התוסף שלנו.
4. סעיף זה הוא אופציונלי – רק במידה ויש קונפיגורציות נוספות לתוסף הספציפי – לכן יש לקרוא טוב את התיעוד עליו ולהבין מה צריך לכתוב שם.
5. בחזרה לבעיה שלנו – אנחנו מבינים עכשיו שבשביל ליצור קובץ html אוטומטית - אנחנו צריכים תוסף כלשהו. בתרגיל זה נשתמש [בתוסף הבא.](https://webpack.js.org/plugins/html-webpack-plugin/) **הוסיפו אותו לפרויקט**.
6. **נריץ את פקודת הבדיקה שלנו** ואם הכל נעשה כראוי – ייווצר קובץ html חדש ובו תגיות שמכוונות לקבצים המאוגדים מהתרגיל הקודם.

**תרגיל #5 – "מנקים את הDist"**

בתרגיל הזה המטרה שלנו זה כל הרצה של הWebpack לנקות את תיקיית Dist מתוכנה. נעשה זאת באמצעות התוסף [הבא.](https://github.com/johnagan/clean-webpack-plugin) בהצלחה!

שימו לב! אתם רק מוסיפים לפתרון של התרגיל הקודם את התוסף ומנסים למחוק את תיקיית dist. לנוחיותכם יצרנו עבורכם תיקייה כזו עם המון קבצים סתמיים ועליכם למחוק אותם. במידה והצלחתם תיווצר תיקייה חדשה בשם dist רק עם התוצאה שציפיתם לב התרגיל הקודם במקום הקבצים הסתמיים.

דגש: מופע חדש של התוסף לא מקבל כלום כפרמטר.