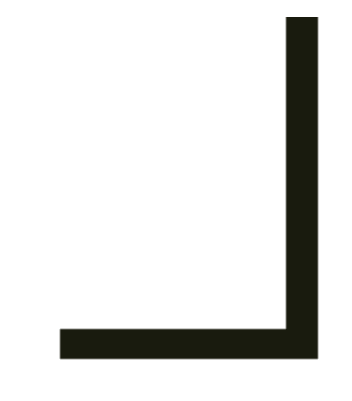


**Webpack**

**Workshop**



**לפני שמתחילים**

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

שימו ♥ - לפני קריאת החוברת הזו, יש לקרוא את החוברת למתחילים!

המטרה של חוברת זו היא להכיר עוד עקרונות ומושגים בעולם ה Webpack ועל הדרך להכיר עוד כמה טריקים לפיתוח.

בנוסף, חשוב לציין שהחוברת הזו היא בעלת ציר סיבוך נוסף שלא היה בחוברת הקודמת, בחוברת זו לא יהיה לכם צעדים של מה שצריך לעשות בכל שאלה – אנחנו נחפש באינטרנט, את מה שאנחנו צריכים ואיך להוסיף אותו לקונפיגורציות שלנו!

אז עכשיו שהכל סגור – בואו נצא לדרך!

**Webpack ככלי פיתוח**

עד עכשיו השתמשנו בWebpack ככלי שנועד בעיקר לחלקים של סוף הפיתוח – כאשר סיימנו לפתח ואנחנו רוצים לייצא ולאגד את כל הקבצים והתלויות שלנו לקובץ אחד וכדומה.

תהליך הפיתוח שלנו בעזרת אותו כלי נראה (בערך) כך:

1. אנחנו כותבים / משכתבים את הקוד.
2. אנחנו מעבירים את הקוד שכתבנו (על גבי מספר קבצים ובשימוש עם ספריות חיצוניות) את תהליך ה"איגוד" של ה Webpack.
3. אנחנו פותחים את התוצרים של התהליך ורואים את התוצאות.
4. חוזר חלילה.

אבל, Webpack זה כלי מאוד ורסטילי וניתן להשתמש בו לכמה מטרות – עכשיו אנחנו נלמד כמה טריקים לפיתוח יותר נוח בעזרת ה Webpack, זכרו שהם רק חלק קטן מהאפשרויות פיתוח ש Webpack מציע לנו, לכן ברגע שאתם נתקלים בבעיה – חפשו אם יש פתרון שהכלי מציע באינטרנט, סביר להניח שיש.

שימו ♥ - הדברים השונים שיוצגו כאן, הם חלק קטן מהמון אפשרויות של Webpack אשר מקלות על הפיתוח, מוזמנים לחקור באינטרנט!

**חזרה קצרה על Scripts**

בתחילת החוברת הקודמת דיברנו על קובץ ה Package.json, שהוא קובץ חדש שנוצר לנו עם הרצת הפקודה הבאה:

npm init

הקובץ מכיל פרטים אודות הפרויקט כגון: היוצרים, גרסה, ספריות שמשומשות בפרויקט ועוד.  
בנוסף, יש לנו בקובץ מאפיין בשם "Scripts" והוא מכיל אובייקט Json שכל מאפיין שלו הוא סקריפט שאנחנו יכולים להריץ ושם המאפיין הוא שם הסקריפט. נריץ את הסקריפט באמצעות הפקודה:

npm run שם סקריפט

בחוברת הקודמת יצרנו סקריפט חדש בשם "test" שמטרתו היא להפעיל את תהליך האיגוד ולאחר מכן אנחנו יכולים להריץ את התוצרים של התהליך ולבדוק האם הכל עבד כראוי.

בתור סקריפט אנחנו יכולים להפעיל מספר פקודות ואף להעביר משתנים שישנו את אופן ההרצה של אותו סקריפט.

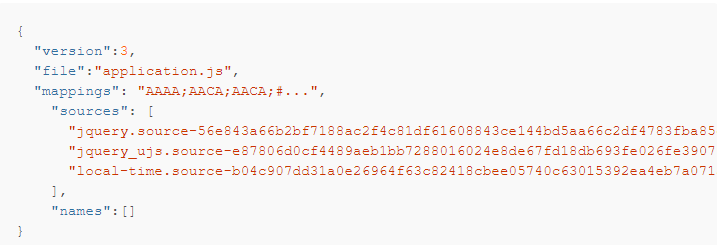
**תרגיל #1 – "על איזה סביבה מדובר?"**

Webpack הכניס את עקרון ה"מארז", בעצם שהקוד שלנו נארז לאחר שאנחנו מסיימים לכתוב אותו (יחד עם כל התלויות והLoaderים השונים) ועל כן – הוא שונה מצורתו המקורית (בסביבת עבודה שלנו) כאשר אנחנו פותחים אותו בדפדפן.

הבעיה המרכזית בכך היא שכאשר אנחנו רוצים לדבג את הקוד, אנחנו נלחץ על F12 (על מנת להיכנס ל DevTools) ויהיה לנו קשה לעבוד ולמצוא את הבאג מאחר והקוד שונה.

אנחנו מבינים שעלינו ליצור הפרדה בין שני מצבים: Development ו Production מאחר ואולי לא נרצה לאפשר ללקוחות שלנו להבין את הקוד שלנו כאשר המערכת שלנו תהיה באוויר ואנחנו גם רוצים את כל היתרונות ש Webpack מציע כמו כיווץ גדלי הקבצים וכדומה.

בשביל לענות על הדרישה שלנו אנחנו נלמד על מושג חדש בשם "Source Map":

Source Maps הם קבצי Json שאנחנו יכולים לייצא באמצעות ה Webpack שמכיל את כל מה שהדפדפן צריך כדי למפות את הקוד ה"ארוז" (כלומר – לאחר ה Webpack) חזרה לקוד הקיים במידה והמתכנת מגדיר לו כך, כלומר הם בסך הכל קבצים שעוזרים לדפדפן לפרש את הקוד ה"ארוז". בואו נראה דוגמא לחלק קטן מאחד:

מה שאנחנו רואים פה (בקצרה) – אובייקט Json שמכיל כמה פרטים על הקובץ אותו הדפדפן קיבל וצריך להמיר אותו חזרה לקבצים המקוריים: שם (file), מחרוזת מקודדת, [מוזמנים לקרוא על הקידוד פה](https://en.wikipedia.org/wiki/Variable-length_quantity), שמראה לדפדפן איך למפות את השורות לאיזה קבצים (mapping) ואיזה בספריות השתמשו במהלך הפיתוח (sources) – ופי כל אלה, הדפדפן יודע לתרגם חזרה את הקוד שלנו לצורתו המקורית.

עכשיו שהכרנו את המושג שעומד לפתור לנו את הבעיה, נותרת השאלה איך אנחנו מורים ל Webpack לייצא קובץ שכזה?

אז למזלנו מגרסה 4 של Webpack – הוא ייצר לנו את הקבצים הללו בצורה אוטומטית במידה ואנחנו אומרים לWebpack שאנחנו במצב Development (במצב Production – הקבצים לא ייוצרו), לכן כל מה שעלינו לעשות הוא להריץ את ה Webpack במצב זה.

בתרגיל זה מה שעליכם לעשות הוא להריץ את הפרויקט שקיבלתם במצב Development ולבקש ממנו ליצור Source map מסוג inline-source-map.

[מוזמנים לחקור פה.](https://www.google.com/)

שימו ♥ - אתם מקבלים פרויקט שלם, עם הגדרות מוכנות ואפילו קובץ Package.json, לכן לפני העבודה יש לעשות npm install.

איך תדעו שסיימתם? :

1. תפתחו את הדפדפן.
2. תלחצו F12 ואז על הטאב "Sources".
3. בצד שמאל נלחץ על "Webpack".
4. שם תנסו למצוא את הקובץ – "pokemon.service.js",  
   אם הוא אותו אחד שישנו בפרויקט – הצלחתם!

**טיפ:** ניתן לשנות את ההגדרות של הWebpack שלכם בקובץ שאנחנו רגילים או דרך דגלים של הפקודה "Webpack" – זכרו איפה אנחנו קוראים לה!

**תרגיל #2 – "חלאס"**

לא נמאס לכם כל פעם שאתם עושים שינוי קטן בקוד – אתם צריכים להריץ שוב את אותה הפקודה כל פעם?

npm run test

לא הייתם רוצים שהWebpack פשוט יאזין לשינויים בקוד שלנו וכל פעם שנשנה תהליך הWebpack יחול אוטומטית?

אז יש לנו משהו טוב לספר לכם – [קראו פה.](https://webpack.js.org/configuration/watch/)

מצורף פרויקט בסיסי מאחד התרגילים של החוברת הקודמת.

שימו ♥ - אתם מקבלים פרויקט שלם, עם הגדרות מוכנות ואפילו קובץ Package.json, לכן לפני העבודה יש לעשות npm install.

בתרגיל זה מה שעליכם לעשות זה:

1. לשנות את ההגדרות של הWebpack שיאזינו לשינויים בקוד שלנו.
2. לבדוק האם כאשר אתם משנים את הקוד – נוצרים קבצים חדשים בתיקיית dist.

דגשים:

1. כאשר אנחנו משנים את ההגדרות של הWebpack עצמו – אין האזנה, עלינו להריץ את הפקודה שוב.
2. כאשר נצליח לגרום לWebpack להאזין – על הCMD בו הרצנו את הפקודה להישאר פתוח.

**תרגיל #3 – "חלאס 2"**

לא נמאס לכם שכל פעם שנוצרים קבצי dist חדשים (גם עם האזנה של ה Webpack) – צריך לסגור את הקבצים הישנים ולפתוח את החדשים בדפדפן?

אז גם לבעיה הזו יש פתרון!

Webpack-dev-server היא ספריית NodeJS כל מטרתה היא להקים שרת קבצים סטטי, כלומר שרת שפשוט מחזיר קבצים לפי הגדרות Webpack. אז מה שאנחנו עומדים לעשות זה להרים שרת Webpack-dev-server שיאזין לשינויים בקוד ויתעדכן באופן אוטומטי!

מה אנחנו צריכים לעשות?

1. נמחק את תוכן הסקריפט "test" שבקובץ ה package.json.   
   שימו ♥ - במידה וישנם דגלים הנלווים לפקודה webpack יש להוסיפם בקובץ webpack.config.js.
2. נתקין את הספרייה Webpack-dev-server – [קראו כאן כיצד.](https://github.com/webpack/webpack-dev-server)
3. בסקריפט (הריק) תחת השם "test" – נקרא לפקודה הפשוטה "webpack-dev-server".
4. בקובץ webpack.config.js נוסיף מאפיין בשם devServer אשר יכיל אובייקט עם מאפיין בשם contentBase עם ערך שמכוון לתיקיית הDist כך:

devServer**:** **{**

contentBase**:** "./dist"

**}**

כך אנחנו אומרים לשרת שלנו לייחצן את הקבצים שבתיקיית הdist, במידה ונרצה לייחצן קבצים אחרים נשים נתיב לתיקייה אחרת.

1. נריץ.

איך נדע שסיימנו?

נשנה את הקוד ואם השינויים התעדכנו באופן אוטומטי – הם יקפצו לנו בדפדפן, בלי צורך לפתוח מחדש את הקובץ, השרת שלנו ירוץ באופן דיפולטי בפורט 8080.

דגשים:

כמו בתרגיל הקודם.

אחת הבעיות שנוצרות עקב השימוש בשרת קבצים שייחצן לנו את הקבצים ויציג לנו הכל בצורה אוטומטית היא שמידה ונרצה לשלוח בקשת API לשרת אחר – הבקשה תישלח לשרת קבצים. בואו נראה דוגמא ונבין את הבעיה יותר טוב:

רץ לנו שרת קבצים ב "localhost:8080" כפי שלמדנו בתרגיל הקודם ושרת המייחצן API ב "localhost:9000", אחד הקבצים שלנו – נניח "controller.js" שולח בקשת HTTP לנתיב "/projects" על מנת לקבל את כל הפרויקטים מהצד שרת,  
הדפדפן ילך ישירות לשרת אשר הביא לו את הקבצים ויבקש את הנתיב API כך "localhost:8080/projects", אך ה API שאנחנו רוצים לפנות אליו רץ בפורט 9000.

ניתן לפתור את הבעיה הזאת בכך שנפנה ל API בכתובתו המלאה, כלומר בבקשת HTTP שלנו נפנה לכתובת המדויקת: "localhost:9000/projects", אך גם כאן אנחנו נתקלים בבעיה – מה יקרה כאשר נשנה את הכתובת / הפורט עליה רץ שרת ה API שלנו / נעבור לסביבת Production – אנחנו נצטרך לעבור בכל בקשה ולשנות את הכתובת לשרת API הרלוונטי.

למזלנו הספרייה “webpack-dev-server” מאוד חזקה ונותנת לנו המון פונקציונליות ויכולות שיקלו על הפיתוח שלנו מאוד. בואו נראה איך אנחנו נפתור את הבעיות הללו באמצעותה.

אנחנו נלמד כרגע על מאפיין חדש באובייקט ה "devServer" (אשר נמצא בקובץ webpack.config.js) ושמו הוא "Proxy", בעזרתו נוכל להגדיר איזה בקשות ינותבו לאן:

בקטע הנ"ל - אנחנו מגדירים שכל בקשה שתתחיל בנתיב "/api" – תנותב לכתובת "localhost:9000".

devServer**:** **{**

proxy**:** **{**

'/api'**:** 'http://localhost:9000'

**}**

אם נחזור לדוגמא הקודמת – הבקשה שאני אשלח הפעם תהיה "/api/projects" ובגלל שהיא מתחילה עם התחילית "api" היא תנותב לכתובת: "localhost:9000/api/projects".

כמובן שבמידה ואנחנו פונים לכמה שרתים, אפשר להוסיף עוד ועוד הגדרות, מוזמנים לקרוא [פה.](https://webpack.js.org/configuration/dev-server/)

**מעכשיו כאשר נרצה לפנות לשרת API מתוך שרת קבצים של Webpack נפנה כך!**

שימו ♥ - ההגדרות הללו תקפות בסביבת הפיתוח בלבד!

**תרגיל #4 – "קיצורי דרך"**

ככל שחולף הזמן – הפרויקט שלנו גדל וגדל וכך גם כמות המודולים והקבצים בו. בסופו של דבר הכמות תהיה כל כך גדולה – שהמעבר בין הקבצים והניווט בכל ה File System של הפרויקט שלך הופך להיות לא- יעיל, גוזל זמן ומועד לטעויות, חשבו רק על ה importים שלכם, המצב ייראה כך:

**import** background **from** '../../../../styles/background.css'**;**

**import** **{** ROUTES **}** **from** '../../../../routes'**;**

**import** **{** setStorage**,** getStorage **}** **from** '../../../config/storage'**;**

**import** **{** setMemory**,** getMemory **}** **from** '../../../config/Memory**;**

הפתרון ש Webpack מציע הוא ליצור כינויים או aliases לנתיבים מסוימים דרך הקובץ webpack.config.js, כדי שנוכל להשתמש בהם במקום להשתמש בנתיבים מסורבלים כאשר אנחנו קוראים ל require או import.

נעשה זאת באמצעות הוספת אובייקט חדש בקובץ הגדרות ה Webpack שלנו בשם "resolve", הוא האובייקט שאחראי בהגדרות ה Webpack לדרך שבה המודולים מתחברים ביחד.  
בתוך אותו אובייקט נוסיף עוד אובייקט בשם "alias" שהוא אחראי לכינויים שניתן לנתיבים השונים ובו בכל מאפיין יהיה נתיב כך:

resolve**:{**

alias**:{**

styles**:** path**.**resolve**(**\_\_dirname**,** 'src/styles/'**),**

routes**:** path**.**resolve**(**\_\_dirname**,** 'src/routes.js'**),**

config**:** path**.**resolve**(**\_\_dirname**,** 'src/config/'**)**

**}**

**}**

בדוגמא הנ"ל אנחנו יוצרים נתיבים מסודרים לתיקיות config ו styles וקיצור דרך ישיר לקובץ routes.js. לאחר מכן נוכל להחליף את כל הimportים בדוגמא למעלה בקטע קוד הבא:

**import** background **from** 'styles/background.css'**;**

**import** **{** ROUTES **}** **from** 'routes'**;**

**import** **{** setStorage**,** getStorage **}** **from** 'config/storage'**;**

**import** **{** setMemory**,** getMemory **}** **from** 'config/storage'**;**

הרבה יותר יפה לא?

שימו ♥ - לקיצורים שעשינו ישנן כמה השלכות ודגשים:

1. את משתנה path יש לייבא באמצעות require. מוזמנים לקרוא עוד על משתנה path [כאן.](https://nodejs.dev/nodejs-file-paths)
2. במידה ואנחנו משתמשים בהרחבות של JS, כדוגמת TypeScript – יש לבדוק האם גם בהגדרות השפה צריך להוסיף הגדרות.
3. מנגנון ההשלמה האוטומטית (IntelliSense) לא יעבוד עם הנתיבים החדשים שלנו – כדי לפתור את הבעיה, יש להוסיף את הנתיבים בקובץ jsconfig.json או להשתמש בהרחבה של Visual Studio Code.

אין תרגיל - מוזמנים להתנסות עם האופציה ולראות איך הקוד שלכם הופך לנקי יותר.

**תרגיל #5 – "באבל"**

נניח ואנחנו רוצים לפתוח את המערכת החדשה שלנו, שכתובה בגרסה האחרונה של JS, על דפדפן מאוד ישן – כנראה שניתקל בבעיות מאחר והדפדפן הישן לא יודע לעבוד עם הגרסאות החדשות.

את הבעיה הזאת אנחנו נפתור בקלות באמצעות כלי חדש שנקרא: "Babel".

Babel הוא כלי שנועד להמיר קוד שנכתב ב ECMAScript 2015 (ומעלה) לגרסאות קודמות כדי שדפדפנים ישנים יוכלו להריץ את המערכת. מוזמנים להמשיך לקרוא [כאן](https://babeljs.io/docs/en/index.html) ולהתנסות עם הכלי [כאן](https://babeljs.io/repl).

אבל, כלי ה Webpack לא מבאבל (מלשון Babel) את הקוד שלנו בצורה אוטומטית וכלן נצטרך להגדיר Loader חדש עבור קבצי JS, כדי שיעברו את ההמרה לסינטקס המיושן.

בתרגיל הזה עליכם להתקין ולקנפג babel-loader, מוזמנים להיעזר בחוברת הקודמת [ובאתר הזה](https://babeljs.io/setup#installation) ובמדריך [הזה.](https://www.thebasement.be/working-with-babel-7-and-webpack/)

לפניכם קוד שכתוב בגרסה מתקדמת של ES, כל מה שהקוד עושה הוא להדפיס התראות של "1", לאחר מכן "2" ואז "3". במידה ותפתחו את המערכת (localhost:8080) בדפדפן Google Chrome – הקוד ירוץ ויעבוד כמצופה, אך במידה ותפתחו אותו בדפדפן Internet Explorer – הקוד לא יעבוד.

עליכם לגרום לקוד לעבוד גם בדפדפן הזה, השתמשו במדריכים ובקישורים כדי לעשות זאת.