

תודה רבה על הרכישה של הספר!

עבדתי מאוד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוכרי העריכאה. יותר מ-1800 אנשים תמכו בספר זהה ואיפשרו לו לצאת לאור.

הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהקינדל, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתוך תקווה שהרוכש והותמן לא ינצל את האמון שנתתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפיצה שלו. אני מאמין שרוב האנשים הוגנים.

העתק זהה נמכר ל:

nivbuskila@icloud.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נჩבים פרטיו הרוכש באופן שקוף למשתמש. כדאי מאד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיהקנו את העתק שנמצא ברשותכם.

תודה וקריאה נעימה!



# ללמוד ריינט בעברית

נן בר-זיך



הקריה האקדמית אונו  
Ono Academic College  
חוג למדעי המחשב

# לימוד ריאקט בעברית

רן בר-זיק

מהדורה: 2.1.0



Really Good



כל הזכויות שמורות © רן בר-זיק, 2021.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רישיון לא-בלודי, לא-ייחודי, אישי, בלתוי נתן להעברה (למעט על פי דין), ובلتוי ניתן להסבה לעשות שימוש אישי בספר זה לצרכים לימודים בלבד.

אסור לך להעתיק את הספר, לשכפל אותו, ליצור יצירות נגזרות ממנו או לפרסם אותו בכל צורה אחרת.

מותר לך לצלט קטעים קצרים מהספר במסגרת הגנת שימוש הוגן, ככלומר פסקה או שתים, כאשר אתה מפנה למקור ומציר את רן בר-זיק כמחבר הספר.

הדוגמאות המובאות בספר זה הן בבעלות של רן בר-זיק, ואסור לך להשתמש בהן בתחום תוכנות שתפתח. אם אתה רוצה להכניס אותן לפרויקט שלך, שלח מייל ונדבר על זה.

עריכה לשונית: יעל ניר  
הגהה: חנן קפלן  
עיצוב הספר והכricaה: טל סולומון ורדי ([tsv.co.il](http://tsv.co.il))

הפקה: כריכה – סוכנות לסופרים

[www.kricha.co.il](http://www.kricha.co.il)



## תוכן העניינים

<b>9 .....</b>	<b>על "לימוד ריאקט בעברית"</b>
<b>10 .....</b>	<b>על המחבר</b>
<b>11 .....</b>	<b>על העורכים הטכניים</b>
<b>11 .....</b>	<b>דורון זבלבסקי</b>
<b>11 .....</b>	<b>gil pinck</b>
<b>12 .....</b>	<b>דורון קילוי</b>
<b>13 .....</b>	<b>על חברות התומכות</b>
<b>13 .....</b>	<b>Really Good</b>
<b>13 .....</b>	<b>אלמנטור</b>
<b>15 .....</b>	<b>HoneyBook</b>
<b>16 .....</b>	<b>על ריאקט</b>
<b>18 .....</b>	<b>דרך הלימוד</b>
<b>18 .....</b>	<b>על המונחים בעברית</b>
<b>19 .....</b>	<b>סביבת העבודה הבסיסית</b>
<b>22 .....</b>	<b>סביבת עבודה בסיסית ב-codepen</b>
<b>25 .....</b>	<b>אפליקציית ריאקט</b>
<b>31 .....</b>	<b>בנייה קומפוננטת פונקצייה בסביבת העבודה הבסיסית</b>
<b>39 .....</b>	<b>בנייה של סביבת ריאקט אמיתית ומורכבת יותר באמצעות Vite</b>
<b>39 .....</b>	<b>התקנת Node.js על המחשב שלכם</b>
<b>40 .....</b>	<b>התקנה על חלונות</b>
<b>43 .....</b>	<b>התקנה על Mac</b>
<b>44 .....</b>	<b>התקנה על Linux</b>
<b>44 .....</b>	<b>עבודה עם טרמינל</b>
<b>44 .....</b>	<b>הפעלת הטרמינל</b>
<b>46 .....</b>	<b>גיוס בטרמינל</b>
<b>47 .....</b>	<b>מציאת מיקומים בטרמינל דרך חלונות</b>
<b>48 .....</b>	<b>טרמינל ב-Visual Studio Code</b>
<b>48 .....</b>	<b>בדיקות גרסות Node.js דרך הטרמינל</b>
<b>49 .....</b>	<b>עבודה עם Vite</b>
<b>54 .....</b>	<b>קשיים ותקלות</b>
<b>54 .....</b>	<b>Create React App</b>

<b>56</b>	<b>כתיבת קומפוננטה ראשונה ב-Vite</b>
<b>65</b>	<b>Export \ Import</b>
<b>68</b>	<b>יבוא מנתיבים אחרים</b>
<b>70</b>	<b>אין חובה להשתמש בסימט הקובץ ts</b>
<b>70</b>	<b>יצוא של כמה משתנים</b>
<b>71</b>	<b>יבוא של תמונות, CSS ומשאים אחרים</b>
<b>74</b>	<b>JSX</b>
<b>80</b>	<b>רשימות ב-JSX</b>
<b>89</b>	<b>קומפוננטה עם תכונות (<i>props</i>)</b>
<b>103</b>	<b>דיבאג</b>
<b>108</b>	<b>סמיין</b>
<b>110</b>	<b>ריאקט ורנדור</b>
<b>110</b>	<b>סטייט</b>
<b>121</b>	<b>תכנון מבנה הקומפוננטות</b>
<b>126</b>	<b>איروسים ועדכון קומפוננטות</b>
<b>127</b>	<b>אירוע DOM</b>
<b>128</b>	<b>איروسים סינטטיים / ריאקטיים</b>
<b>149</b>	<b>אלמנט <i>fragment</i></b>
<b>153</b>	<b>useEffects</b>
<b>159</b>	<b>קומפוננטה ללא שם</b>
<b>162</b>	<b>עיצוב קומפוננטות</b>
<b>164</b>	<b>CSS בסיסי</b>
<b>165</b>	<b>סלקטו</b>
<b>167</b>	<b>תבונות</b>
<b>171</b>	<b>קלאס ריאקטי</b>
<b>181</b>	<b>מעגל החיים בקלאס</b>
<b>181</b>	<b>componentDidMount</b>
<b>181</b>	<b>componentDidUpdate (prevProps, prevState, snapshot)</b>
<b>181</b>	<b>componentWillUnmount</b>
<b>181</b>	<b>shouldComponentUpdate (nextProps, nextState)</b>
<b>182</b>	<b>getDerivedStateFromError (error)</b>
<b>182</b>	<b>componentDidCatch (error, info)</b>

<b>192.....</b>	<b>שימוש בקומפוננטות מקורות אחרים.....</b>
<b>197.....</b>	<b>"יבוא כמו קומפוננטות.....</b>
<b>199.....</b>	<b>מודולים חסרים.....</b>
<b>212.....</b>	<b>HOC: Higher Order Component</b>
<b>225.....</b>	<b>ראונינג.....</b>
<b>247.....</b>	<b>קונטנסט.....</b>
<b>262.....</b>	<b>חיבור לשרת עם סרוויסים.....</b>
<b>270.....</b>	<b>מבוא לבדיקות עם Vitest.....</b>
271.....	התקנת vitest
<b>273.....</b>	<b>חלק ראשון – import –</b>
<b>273.....</b>	<b>חלק שני – כתיבת מסגרת הבדיקה.....</b>
<b>273.....</b>	<b>חלק שלישי – הרצת הקומפוננטה.....</b>
<b>273.....</b>	<b>חלק רביעי – הבדיקה.....</b>
<b>276.....</b>	<b>בדיקות לקומפוננטה עם props .....</b>
277.....	not
<b>280.....</b>	<b>יצירת בילד והעלאת האפליקציה לサーバ חיה.....</b>
<b>286.....</b>	<b>סיכום – ומה עכשו?.....</b>
<b>287.....</b>	<b>התחברות להקהilit הפיתוח.....</b>
<b>287.....</b>	<b>פגשים ומיטאפים.....</b>
<b>287.....</b>	<b>האתר Stackoverflow.....</b>
<b>288.....</b>	<b>תרומות קוד בגייטהאב.....</b>
<b>289.....</b>	<b>נספח: PropType.....</b>
<b>289.....</b>	<b>התקנת PropTypes.....</b>
<b>290.....</b>	<b>השימוש ב-PropTypes.....</b>
<b>294.....</b>	<b>ערכים שאפשר לקבוע.....</b>
<b>296.....</b>	<b>נספח: שינויים מהמהדורה הקודמת (מהדורה 2.0) .....</b>
<b>296.....</b>	<b>נספח: שינויים מהמהדורה הקודמת (מהדורה 1.1.1) .....</b>
<b>296.....</b>	<b>נספח: שינויים בין מהדורה 1.1.1 למהדורה 1.0.0 .....</b>

# על "לימוד ריאקט בעברית"

**ריאקט** היא ספרייה פופולרית מאוד לפיתוח אתרי אינטרנט, אפליקציות ווב, ובכלל כל סוג של ממשק אינטרנט. באמצעות ריאקט והאקויסיטם הנרחב והעשיר שלו, כולל קומפוננטות רבות המלוות אותה, כל מתכנת יכול ליצור אתר או אפליקציה מורכבים במהירות רבה ובאיכות יוצאת דופן. לריאקט יש גם מעתפת (כמו אלקטرون או קורדובה) המאפשרת ממשק שפותח לוויב להיות מותאם בקלות גם לתוכנות מחשב של ממש. העושר של האקויסיטם הפיתוחי של ריאקט וחוזק הקהילה שלו בארץ ובעולם מאפשרים לקבל גם המון-המון מידע וסיווע בכל תקלה ובעיה. אם מתכנת נתקל בבעיה בפיתוח, סביר להניח שהוא ימצא סיוע ומידע בפורומים ובקבוצות מתכנתים. יש גם שפע של מיטאים וקבוצות דיוון המוקדשים למתקנים העובדים בריאקט ונוסף על כן, הביקוש למתקנים המכירים את ריאקט הוא גבוה. כל הדברים הללו הופכים את ריאקט לאידיאלית לכינסה לעולם פיתוח צד הלקוח.

הספר מניח ידע מקיף בג'אווהסקריפט, טייפסקריפט וידע בסיסי ב-HTML וב-CSS. אם איןכם מכירים ג'אווהסקריפט וטייפסקריפט, אפשר ללמוד את שני הנושאים בספר "לימוד ג'אווהסקריפט בעברית". יש שם פרק המסביר על HTML בסיסי ועל CSS בסיסי. מי שישים לקרוא את הספר ההוא ותרגל נהלה, אמור להחזיק בידע מספק על מנת להתחיל ללימוד ריאקט. בספר זה נלמד על ריאקט ממש מהבסיס – הכרת מונחים בסיסיים ויצירת סביבת עבודה – ונגיע עד חומרים מתקדמים כמו state hooks high order components ו בשפת טייפסקריפט המודרנית. הספר מעודכן לגרסה ריאקט 18 ול-Vite.

## על המחבר

REN BAR-ZIK הוא מפתח תוכנה במגוון שפות ופלטפורמות מאז 1996 ועובד כמפתח בכיר במרכז פיתוח של חברות רב-לאומיות, מ-HPE ועד Verizon, שם הוא מפתח בטכניקות מתקדמות הן בצד הליקוח, הן בצד השירות, ושם דגש על בניית תשתיות פיתוח נכונה, על שימוש ב-CD\CI וכמו כן על אבטחת מידע.

נוסף על עבודתו כמפתח במשרחה מלאה, REN הוא עיתונאי ב"זה מרקר" במדור המחשבים, שם הוא מסקר נושאים הקשורים לטכנולוגיה ולאבטחת מידע וכותב על אינטרנט ורשתות.

משנת 2008 מפעיל REN את האתר "אינטרנט ישראל" (internet-israel.com), שהוא אתר טכני המכיל מדריכים, מאמרים והסבירים על תכונות בעברית ומתקדם לפחות פעם בשבוע.

REN הוא מחבר הספר "לימוד ג'אווה סקריפט בעברית", "לימוד Node.js בעברית", "לימוד MySQL בעברית" והספר "לימוד פיתוח ווב מעשי בעברית" ומלמד בקרייה האקדמית אוננו.

REN נשוי ליעל ואב לארבעה ילדים: עומר, כפיר, דניאל ומיכל. רץ למרחקים ארוכים וחובב טולקין מושבע.

# על העורכים הטכניים

## דורון זבלבסקי

הקריירה של דורון התחילה דואק באצד השרת, בחברות אבטחת מידע שבהן עבד כמתנדס תוכנה, מוביל טכני ומנהל מוצר. את המעבר לעולם הפורנטאנד ביצע בשנת 2014 כאשר הצטרף ל-Appplitools, שם הקים את קבוצת פיתוח הוב, וכיום הוא מוביל אותה. במסגרת חיפושים וניסיונות להחיל עקרונות נכונים של הנדסת תוכנה על קוד פורנטאנד הוא התווודע לריאקט ואמץ אותה בחום. דורון הקים את קהילת ריאקט בישראל, כולל קבוצת פיסבוק ומיטאפ פופולרי, ואף יוזם את כנס ReactNext הראשון בישראל והוא מפיק משותף שלו. בנוסף לכך הוא מרצה על ריאקט, על בדיקות אוטומטיות ועל בניית צוותים מנכחים ותורם מזמנו בשמחה למתחילים המבקשים סיוע בתחום במסגרת מפגשים אישיים וקובוצתיים.

## gil Fink

gil Fink הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert .sparXys Microsoft Developer Technologies MVP. כיום הוא מייעץ לחברות ולארגוני שונים, שם הוא מסייע בפיתוח פתרונות מבוססי אינטרנט SPA. הוא עורך הרצאות וסדנאות לייחדים לחברות המוניניות להתחמות בתשתיות, בארכיטקטורה ובפיתוח של מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט Pro Single Page Application (Microsoft Official Course MOC), מחבר משותף של הספר "Development AngularUP (Apress) ושותף בארגון הכנס הבינלאומי UP".

לפרטים נוספים על gil: <http://www.gilfink.net>

## דורון קילזי

דורון חי ונושם פיתוח לרשת זה כעשור. את דרכו החל ביחידה טכנולוגית מובחרת בחיל המודיעין, שבה שירת כSSH שנים. במהלך שירותו הקים ופיתח מערכות מבצעיות מורכבות והיה אחראי על הטמעה של טכנולוגיות חדשות. בארבע השנים האחרונות דורון מפתח full-stack ב-*Verizon Media*, אחת חברות האינטרנט הגדולות בעולם. בימים אלו הוא עובק, בין היתר, במעבר של החברה מטכנולוגיות ותיקות כגון *Angular.js* לריאקט ובפיתוח של מערכות מתקדמות בעולם ה-*ad-tech*. הדחף של דорון ללמידה ולהשתפר גורם לו להמשיך להתקצע ולהתאהב בכל יום חדש בעולם ה-*js*.

# על החברות התומכות

## Really Good

Really Good היא בוטיק פיתוח Front End שעובדת עם סטארטאפים וחברות טכנולוגיות מאז הקמתה ב-2012 על ידי שחר טל ורוני אורבך. אנחנו נוהנים לבנות אפליקציות מורכבות עם UX מוקפם במגוון טכנולוגיות ללקוחות מעוניינים שחוויות המשמש חשוב להם, ושומרים על איזון בריא בין עבודה לחיים.

אנחנו מגייסים מפתחי Front End מנוסים וממש טובים עם תשומת לב לפרטים הקטנים.

[ReallyGood.co.il](http://ReallyGood.co.il)

## אלמנטור

אלמנטור מפתחת פלטפורמת קוד פתוח לבניית אתרים שימושה את הדרך בה בונים אתרי אינטרנט בשוק המkteבי. אלמנטור מעניק למשתמשים את החופש ליצור עמודי אינטרנט ללא צורך בקוד ולפתחים את החירות לדחוף את הגבולות, לרענן ולהרחיב את המערכת בצורה קלה ומהירה באמצעות API ייחודי למפתחים, ובכך לחסוך זמן פיתוח ולהוות יעילים ורוחניים.

עם מיליון+)\ אתרים הפעילים על אלמנטור וצמיחה חודשית מדיהימה, התגבשה סביבה פלטפורמה קהילתית חזקה המונה מאות אלפי חברים, מפתחים, מושוקים ומעצבים, המקייםים מיטאים בכל רחבי העולם. מידי יום האלמנטוריסטים מייצרים וצורכים אלפי שעות של הדרכות, סרטים השראות ובלוגים עמוקים, ומפתחים תורמים קוד ורעיון נאות באמצעות GitHub. האקויסיסטם המkteבי של אלמנטור מתפתח ללא הפסקה והוא אוצר המוסף ומעשיר את יכולות של כל יוצר אינטרנט.

באלמנטור אנחנו משתמשים בטכנולוגיות קוד פתוח מתקדמות לפיתוח כל-אינטרנט חדשניים ו מהירים. אם גם אתם רוצים להיות חלק מהטכנולוגיה שמשנה את חוות האינטרנט בעולם ויש לכם את הידע כדי לבנות עולם יפה יותר אנחנו מתחשים אתכם, מעצבים UI&UX, מפתחים Full

לימוד ריאקט בעברית למד ריאקט בעברית – רן בר-זיק  
הנדסי DevOps ו Big Data Stack  
AWS & GCP.

אתר החברה: <https://elementor.com>

עמוד המשרות: <https://careers.elementor.com/>

## HoneyBook

חברת HoneyBook מפתחת פלטפורמה לניהול פיננסי ועסקי עבור עצמאיים ועסקים קטנים. החברה מאפשרת לקוחותיה לנהל את כל הלידים בצורה אפקטיבית יותר, ניהול כל התקשורת מול לקוחות הקצה שלהם, ניהול כספים והעברת תשלום, חתימת חוזים, תזמון פגישות, ניהול משימות, אוטומציה וכו'.

הפלטפורמה עוזרת יומ-יום לעשרות אלפי אנשים בארץ"ב להתנהל בצורה אפקטיבית ומקצועית יותר, כך שהם יכולים יותר עסקאות בפחות זמן ומאז. את הזמן הפנוי שלהם הם יכולים להשקיע בהגדלת העסק, מציאת עוד לקוחות ובmpsחה שלהם.

בהאניבוק הלקוח הוא המרכז ואיתו גם הbranding שלו. חשוב לנו לוודא שאנחנו מאפשרים לו להראות היכי טוב שהוא יכול בתקשורת מול לקוחותיו שלו. עם טכנולוגיות מתקדמות ודגש על עיצוב, אנחנו מאפשרים לו ביכולות לבנות חוזים, הצעות מחיר ואיימילים שנראים טוב ומאפשרים תקשורת מהירה ויעילה עם לקוחותיו שלו.

אנחנו מתמודדים עם אתגרים טכנולוגיים, עיצוביים ופיננסיים. כאשר בכל אתגר אנחנו שמים את הלקוח במרכז על מנת להגיע להחלטה נכונה ומהירה. והוא תctrpol לחברה מצילה שרצה לשנות את הדרך בה עצמאיים עושים עסקים. חברה שמאפשרת ללקוחותיה להתפרנס ממלחמות שלהם. וכבר הזכרנו שהופענו ברשימה העבודה האטרקטיבית ביותר לשנת 2018 ו-2019? גם בישראל וגם בסן פרנסיסקו (אם תהיתם).

אז למה לעבוד בהאניבוק? כי התרבות העבודה פה מדינה. כי כדי להגיע כל בוקר לעבוד עם אנשים מוכשרים כל כך. כי כל יום שומעים מאות פידבקים מדינים מלקוחות שונים להם את החיים. כי האתגר הטכנולוגי דוחף אותנו כל יום לבנות דברים חדשים ולשפר את מה שכבר בנו.

עמוד המשרות שלנו: <https://www.honeybook.com/careers>

# על ריאקט

צד הלקוח הוא הכינוי לקבצים ששרת האינטרנט שולח אל המשתמש והם-הם בעצם אתר האינטרנט. בדרך כלל מדובר בקובץ אחד או יותר של HTML ו-CSS, בקובצי תמונות ובקובצי ג'אווסקריפט. הדף יודע לקרוא את כל הקבצים האלה ולבנות מהם תמונה של אתר, ה-HTML קובע את מבנה האתר, ה-CSS והתמצאות קובעים את העיצוב שלו וקובצי ה-HTML קובע את התנהוגתו. בתחילת ימי הרשת כך נראה אתר אינטרנט; בתחילת הג'אווסקריפט את התנהוגתו. בתחילת ימי הרשת כך נראה אתר אינטרנט; בתחילת הג'אווסקריפט שימשה לאיניציות או לאינדייציות שונות בדף ובמהמשך לתקדים מתחכמים יותר כמו שימוש בקשות באמצעות AJAX. ובכל זאת, באתר האינטרנט המקורי, כל אינטרנט אקדמי ניוטן כלשהו – לחיצה על כפתור בתפריט או שיגור טופס – שירה בקשה לשרת וגרמה לטעינה מחודשת שלו בידי הדף ולקבלת סט חדש של קובצי HTML, CSS וג'אווסקריפט.

במהלך הזמן החלו להתפתח ספריות ג'אווסקריפט, כמו jQuery. ספריית jQuery הייתה ספרייה עזר שסייעת לתוכני ג'אווסקריפט ליצור אפקטים ואנייציות בклות הרבה יותר. קוד הג'אווסקריפט הצד הלקוח הפך להיות משמעתי יותר ויותר. בשנת 2008 יצא לשוק ספריה שנקראת Backbone.js. זו הייתה ספרייה ששינתה חלוטין את הדרך שבה אנו משתמשים לצד הלקוח: במקום קוד שמנדר רק התנהוגות – אפליקציה שלמה שיושבת לצד הלקוח ומתחנגת כמו אפליקציה לצד שרת, כולל אפשרות ניוטן בעמודים, הבאת מידע מ-API של שירותים ועוד שימושים רבים. היתרונות של שימוש בספריות גדולות כאלה, או יותר נכון פרימורקים המגדירים את התנהוגות לצד הלקוח, היו רבים – מקומות פיתוח ועד חווית שימוש יוצאת דופן עבור הלקוח. הפרימורקים האלה אפשרו ליצור SPA – Single Page Application – קלומר, כונגלושים באתר ובמעבר בין דפים אין טעינה מחדש מהשרת אלא מעבר חלק בין דף לדף באמצעות רכיב מבוסס ג'אווסקריפט שנקרא ראוטר (ועליו נלמד בהמשך הספר). Backbone.js הייתה הראשמה, אבל מהר מאוד הגיעו ספריות נוספות כמו Ember.js ו콤בו אングולר. ריאקט הגיע אחרי אングולר וגרסתה הראשמה יצא ביוני 2013. מאז היא תפסה תאוצה ממשמעותית והפכה לאחת מהספריות הגדולות והנפוצות בעולם.

ריект (React) היא ספרייה מבוססת ג'אווסקריפט המיועדת לצד הלקוח. היא מאפשרת לנו ליצור אתרים שלמים ומערכות שלמות בклות רבה ובדרכ מודרנית ו פשוטה. באמצעות ריאקט אנו

יכולים ליצור דפי אינטרנט או אפליקציות שרצות על טלפונים ניידים ואפיו על מחשבים בקלות רבה.

במקור, המפתחת של ריאקט הייתה חברת פיסבוק, שעדיין היא התומכת הראשית שלה והפתחים שלה מוביילים את פיתוח הליבה של ריאקט ומתווים את הדרך. ריאקט מפותחת בראשון קוד פתוח מלא (החל מגרסת 16) וקוד המקור שלה נמצא בGITHub. אפשר להשתמש בה לכל שימוש בצורה חופשית. אחד היתרונות הגדולים בריאקט הוא עשר הקומפוננטות שימושísticas בה, מה שאומר שאפשר ליצור בקלות אפליקציות מורכבות באמצעות שפע הקומפוננטות שפותחים אחרים פיתחו – דבר המאפשר לכל צוות פיתוח שבוחר בריאקט גמישות ועובדת מהירה מאוד. גם סביבת הפיתוח של ריאקט וכל הבדיקות שלה, החינניות לפיתוח בקנה מידה גדול, הם מצוינים ועמידים מאוד. יש לה כמובן גם חסרונות – החיסרון העיקרי הוא שריקט לאקובעת עבור המפתח את הרכיבים שאתם הוא יכול לעבוד, דבר שעלול להוביל לבלבול או לקבלת החלטות לא נכונות, אבל יש מפתחים שיראו זהה יתרון. כך או כך, נכון לזמן כתיבת הספר, רוב המתכננים שצרכיהם לפתח אתר כלשהו לצד הלקוח בוחרים בריאקט.

בשנים האחרונות, טיפסקריפט היא הבחירה הראשונה בכל הנוגע לשימוש בריאקט. היכולת של טיפסקריפט למנוע תקלות שונות הקשורות לסוגי מידע וקלות השימוש בה גרמה לטיפסקריפט להיות נפוצה מאוד. זו הסיבה שבספר אנו לומדים ריאקט עם טיפסקריפט, למרות שניתן לנכון ריאקט עם ג'אוסקריפט בלבד.

הkonsepzia של ריאקט ודרכו העבודה בה לא שונות מהותית מספריות אחרות כמו אנגולר או Vue. אך אם איןכם מכירים אף פרימורק או ספריה של ג'אוסקריפט, ריאקט היא מקום מצוין להתחיל בו את המסע לעולם המופלא של פיתוח צד לקוח. זאת אף שריקט שונה במידה דריכים מהותיות, שאוון נלמד בהמשך, מספריות אחרות כמו אנגולר.

# דרך הלימוד

הניסיונו שלי ללמד שכל דבר חדש בתכנות לומדים דרך הידים. אני ממליץ מאוד להעתיק כל דוגמה וכל קטע קוד בספר, להדביך אותו ב-IDE החביב עליו – כמו Visual Studio Code למשל, לשחק בהם ולבדוק איך הם עובדים. בסוף כל פרק יש תרגילים – אין לי די מילימס כדי להבהיר עד כמה חשוב לפתור אותם ולשבור עליהם את הראש לפני שמציצים בפתרונות ובהסבריהם. כדאי מאוד לא לוותר ולא להרפות ולנסות שוב ושוב עד שבמединים את הפתרון.

יכול להיות שלמרות ההסבירים ולמרות הדוגמאות לא תבינו נושא מסוים או שלא תבינו אותו עד הסוף, לעומק. זה קורה לטוביים ולمبرיקרים ביוטר. הפתרון? חיפוש בגוגל – במיוחד באנגלית. כיוון שריאקט היא כל כך פופולרית, יש סיכוי סביר כי יותר שמשהו כבר נתקל בבעיה זו וכותב עליה משהו. אתרים כמו StackOverflow והפורומים השונים מכילים שפע של מידע ותשובות לשאלות שונות. בנוסף על כן, בפייסבוק יש לא מעט קבוצות מקצועיות בעברית שימושו לשיער לכם – בפרק הסיכון של הספר יש כמה קישורים רלוונטיים.

כמובן שדרך טובה מאוד למדוד היא באמצעות בינה מלאכותית, כמו למשל צ'אט GPT. הבינה המלאכותית הזו זמינה באתר <https://chat.openai.com> וניתן, באמצעות תיבת השיח שם, לשאול שם שאלות או להדביך קוד שלם או שגיאות שונות ולשאול את הבינה המלאכותית מה מקורן ואייך לפתרו אותן.

למרות הפיתוי העז, אני ממליץ לא להעזר יותר מדי בבינה מלאכותית לכתיבת קוד מורכב אלא להתאמץ ולכתוב את הקוד בעצמכם. בעבודה היומיומית עם בינה מלאכותית יש חשיבות רבה לידע בקוד המאפשר גם לבדוק את הקוד המוצע על ידי הבינה המלאכותית וגם להנחות את הבינה המלאכותית בвиיזוע משימות שונות. ניתן לרכוש את הידע הזה רק בעבודה עצמאית.

## על המונחים בעברית

אני כותב בעברית על טכנולוגיה ועל תכנות כבר יותר מעשור, והדילמה באילו מונחים בעברית להשתמש מלואה אוטי תמיד. מצד אחד, האקדמיה ללשון העברית מספקת לנו מונחים רבים בעברית. מצד שני, בתעשייה ההייטק, שמננה אני מגיע, איש לא משתמש ברבים מהמונחים האלה. אם הגיעו לריאיון עבודה ותגידו: "במפגש המתכנתים האחרון שמעתי על דרך חדשה לבצע

הידור שבודק הזרחות במנשך מבוסס הבטחות", סביר להניח שלא קיבלו את העבודה. אבל אם תגידו, "במיוחד האخرון שמעתי על דרך חדשה לבצע קמפיין שבודק אינדנטציה ב-*API* מבוסס פרומיסום" – יבינו על מה אתם מדברים. זו הסיבה שלא תמצאו בספר מילים כמו "הידור", "מחלקה" או "מרשתת", אלא "קמפיין", "קלאס" ו"אינטרנט". המונחים שבהם השתמשתי הם המונחים שבהם משתמשים בתעשייה בפועל. בכל מקום שבו אני משתמש במונח לראשונה, אני מספק גם את הגרסה שלו באנגלית כדי שתוכלו להכניס אותו לחיפושים שלכם בגוגל או בצ'אט .GPT

חשוב לציין שאיני בז כלל לאקדמיה ללשון וshall ממהונחים שלא אכן נכנסו לשפה המדוברת במרכז הטכנולוגיה השונים (למשל: קובץ או מסד נתונים), אבל בכל מקום שהיתה בידי הבחירה בין להיות מובן לבון לעמוד בכללי הלשון, העדפתו להיות מובן.

## סביבת העבודה הבסיסית

זה הפרק החשוב ביותר, כיוןuai-אפשר למדוד קוד בלי לכלול את הידים בקוד משלכם. קריטי לקרוא את הפרק הזה וליצור סביבת עבודה בסיסית על המחשב שלכם. אם אתם נתקלים בבעיות כאן – אל תוותו ואל תתייחסו. התקנת סביבת עבודה היא החלק הקשה ביותר בלימוד טכנולוגיה חדשה. נסו שוב ושוב – בצעו ריסטרט למחשב, נסו מחשב אחר, שדרגו את מערכת הפעלה, נסו מדף אחר שאינו הדפדפן הרגיל שלכם – כל טכנית וטכנית. פשוט ai-אפשר לדלג על השלב הזה. נסו להעזר בכללי בינה מלאכותית ולהזין לתוכם את השגיאות, אם יש כאלה, כדי לקבל סיוע. זה החלק החשוב ביותר.

ריאקט מורכבת מכמה חלקים – כולם בקוד ג'אווסקריפט. החלק הראשון הוא הספרייה עצמה: ריאקט. מזכיר בקובץ ג'אווסקריפט מרכזי המכיל את כל הפקנציונליות של הספרייה. הקובץ מכיל מודולים של ג'אווסקריפט וגם משתנים גלובליים ואחרים – ובludeio ai-אפשר להשתמש בריאקט. כשאנו בונים סביבת עבודה בסיסית, אנו זקוקים לריאקט. הגיוני, לא? החלק השני הוא *dom-react*. גם הוא חלק מהספרייה וגם הוא כתוב בג'אווסקריפט. הוא מכיל את הפקנציות הקשורות בין ריאקט ל-DOM. ראשית התיבות של DOM הם Document Object Model, ואני ארכח לגביו בהמשך. כרגע צריך פשוט לזכור שמדובר בעוד חלק של הפרימיום רק שהוא צרכיים איתנו כשהאנחנו מפתחים עבור אתרי אינטרנט.

החלק השלישי הוא babel. מה זה? מדובר בספרייה ג'אווהסקרייפט שימושית ופופולרית מאוד. יש לה כמה תפקידים חשובים. במקור babel סייעה למפתחים שרצו להתאים את קוד הג'אווהסקרייפט שלהם לדפדףים יישנים שלא תמכו בפיצרים החדשניים של השפה, למשל דפדףים שלא ידעו מה זה `let` או `const`. הספרייה הזאתלקח את כל הקוד המודרני של ג'אווהסקרייפט והעבירה אותו לתהילין, שבמסגרתו הוא הפק לקוד שתואם גם לדפדףים יישנים. למשל, היא המירה את `let -var`, כך שדף יישן יוכל לעבוד איתו. התהילין הזה נקרא "טרנספירציה" – זו המילה המדעית המחשב שמשמעה לחתך קוד כתוב בשפה מסוימת ותרגם אותו לקוד כתוב בשפה אחרת. במקרה הזה לחתך קוד כתוב בג'אווהסקרייפט מודרנית ולהעביר אותו לקוד כתוב בג'אווהסקרייפט מגרסאות קודמות.

במקרה שלנו, ל-babel יש תפקיד משמעותי בהמרת ה-JS שלנו, שהוא הסינטקס שבו אנו כותבים בריект קומפוננטות לקוד HTML שהדף יודע לעבוד איתו. על JSX נלמד בהרחבה בהמשך.

אלו החלקים הבסיסיים שחיברים להיות בסביבת העבודה שלנו. בנוסף על הקוד שלנו, אנו צריכים ליצור דף HTML ריק לחולtin שקורא באמצעות `src` לשלוות הקבצים האלו וגם לקובץ שבו אנו כותבים את הקוד שלנו.

אנו יכולים להוריד את הקבצים האלו מהאתר של ריאקט או להשתמש בהם [ישירות-cdn](#). ראשי התיבות של CDN הם Content Delivery Network – זהו כינוי לשירותים גדולים שנמצאים בכל מקום בעולם. חלק מהם מציעים שירות פתוח לציבור של אחסון קבצים של ספירות גדולות ומוגן שריאקט, הפריימורק הפופולרי בעולם, נמצא ביןיהם. ה-CDN שהוזקםנטציה של ריאקט משתמש בו והוא `pkg` וANO משתמש בו. `pkg` מוציאה את הקבצים בקישורים הבאים:

קובץ פריימורק של ריאקט:

<https://unpkg.com/react@18/umd/react.development.js>

שימוש לב: הספר מלמד על גרסה 18 של ריאקט ולכן קיים המספר הזה ב-URL.

:react-dom

<https://unpkg.com/react-dom@18/umd/react-dom.development.js>

:babel:

<https://unpkg.com/@babel/standalone/babel.min.js>

כל קובץ ה-HTML שלנו נראה:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
  <script crossorigin
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
</head>
<body>
  <div id="content"></div>
</body>
<script type="text/babel">
  console.log('Here will be React code');
</script>
</html>
```

בתחתית הקובץ נמצא הקוד שלנו. אנו מסמנים לו babel שהוא קוד שצריך לעבר טרנספילציה באמצעות הצמדה של:

`type="text/babel"`

لتגית ה-`script`. בין התגיות הפתוחת של ה-`script` לתגית הסוגרת שלו, אנו יכולים לכתוב את קוד הג'אוועסקריפט שלנו. כרגע יש שם את השורה הבאה:

`console.log('Here will be React code');`

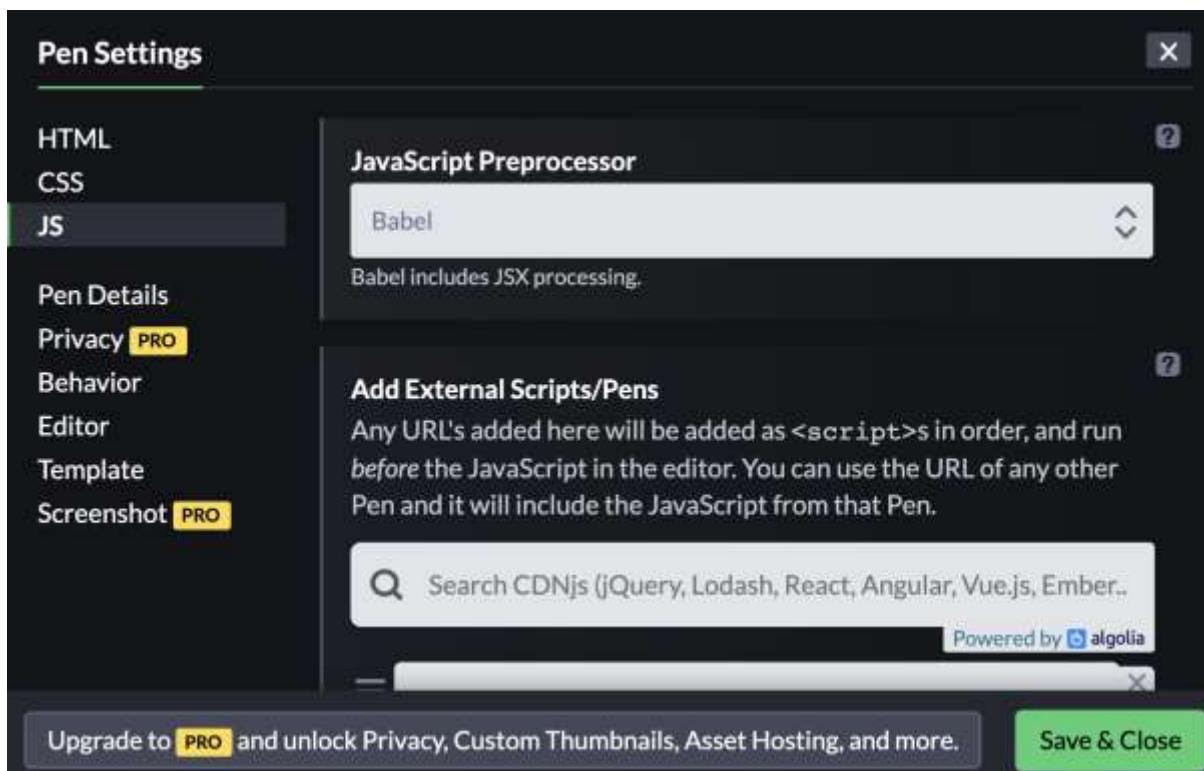
באמצעות עורך הקוד החביב עליוכם (כמו Visual Studio Code החינמי), צרו את קובץ ה-HTML זהה ושמרו אותו במחשב המקומי שלכם. פתחו את קובץ ה-HTML בדף במכשיר באמצעות כניסה לתקייה שבה הקובץ נשמר ולהיצחה על "פתח עם דף פון" ויזדאו שהוא עולה וشبוקונסולה של כל המפתחים מופיע `Here will be React code` כאן – זהו, אנחנו מוכנים לעבודה ראשונית.

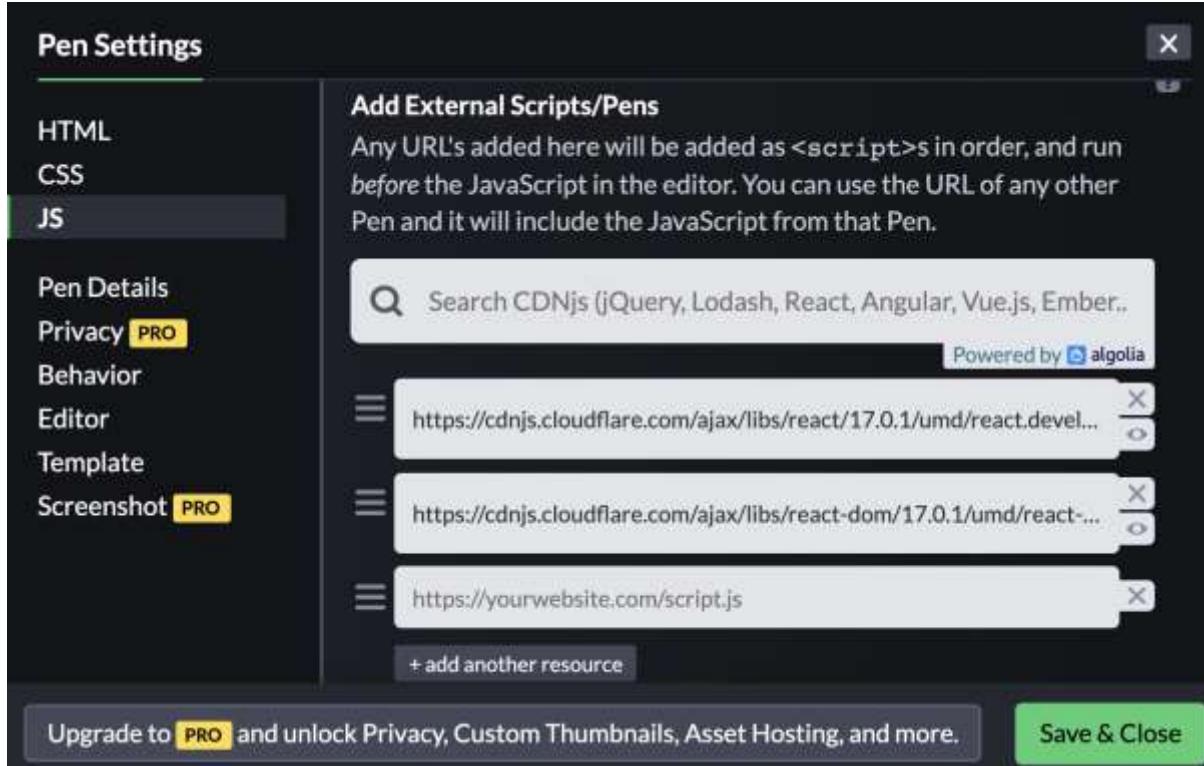
כדי לשימוש לב שאנו עדים לא משלבים טייפסקריפט בקוד, אנו נשלב אותו בסביבת העבודה המתקדמת. כרגע אנו רק לומדים איך ריאקט עובד ולא רוצים לעرب טייפסקריפט.

## סביבה עבודה בסיסית ב-codepen

אפשרות נוספת היא לבנות סביבת עבודה מרוחקת, שגם בה אפשר לכתוב ג'אווהסקריפט ולצפות בתוצאות באמצעות דף-דף בלבד. סביבת העבודה זו זמינה בחינוך בכמה וכמה כלים וארטירים, אבל האתר הכי פופולרי ומוצלח הוא [codepen.io](https://codepen.io/). מדובר באתר שמאפשר לנוכם (אפיו!) בילוי רישום, אף על פי שモולץ להירשם כי זה מאפשר שמירה של הקוד שלכם) לכתוב קוד פשוט בג'אווהסקריפט, CSS ו-HTML. העבודה באתר פשוטה: נכנסים אל <https://codepen.io/pen/>, מתחילהים לכתוב קוד ורואים את התוצאות.

כדי לעבוד עם codepen וריאקט חיבים להכנס את שלושתקובצי הג'אווהסקריפט שהזכרתי קודם: קובץ הpriymowrk של ריאקט, קובץ react-dom וקובץ babel. איך עושים את זה? בהגדרת כל "פרויקט" (שנקרא pen באוטו אחר) הקפידו להכניס את C-preprocessor ו קישור ידני אל קובצי הג'אווהסקריפט מה-CDN. זה נראה כך:





עשיתי את זה עבורכם בפרויקט משמי ואתם יכולים, במקרה לטפל בהגדרות בעצמכם, להיכנס אל הקישור הזה שבו הן מוכנות:

<https://codepen.io/barzik-the-vuer/pen/gOYMWoP>  
צפו בקונסולה וראו את המסר Here will be React code.

חשוב לציין שזו סביבת פיתוח בסיסית מאוד של ריאקט ושהיא כמובן מוכוונת פיתוח בלבד ומיועדת ללימוד, אבל זה אמור להספיק לכתיבת אפליקציות הריאקט שלנו והקומפוננטה הראשונה.

פרק 1

# אפולו הצעית וריאקט



# אפליקציית ריאקט

כל הקוד של ריאקט אמור להיות בתוך אפליקציית ריאקט. זה סוג של מתחם שבו הקוד מבוסס הריאקט שלנו עובד. בעצם, מדובר באלמנט אב של DOM שמתוחתו ריאקט שולטת, יוצרת DOM משלה, שנקרא DOM Virtual, וחייב. יכולות להיות כמה אפליקציות ריאקט בדף HTML אחד. מובן שכל אחת מהן תהיה מתחת לאלמנט DOM אחר. אפליקציית הריאקט היא בעצם אלמנט DOM שאנו מכירזים עליו כשלנו, ובו אנו יכולים ליצור את המרכיבים של האתר שלנו – שם קומפוננטות הריאקט.

כדי ליצור אפליקציה ריאקט מתחת לאלמנט מסוים אנו חייבים פשוט... לבחור אותו. אנו נגידיר div, שהוא אלמנט HTML פשוט ביותר עם id פשוט. להזיכרכם – id הוא סלקטור ייחודי המגדיר אלמנטים ב-HTML שאנו בוחרים מתחת להם זהות ספציפית.  
בקובץ ה-HTML שלנו כבר יש הגדרה של id כזה:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
  <script crossorigin
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
</head>
<body>
  <div id="content"></div>
</body>
<script type="text/babel"></script>
</html>
```

הינה ההגדרה – div שה-id שלו הוא content

```
<div id="content"></div>
```

זה המקום שבו תוצב אפליקציית הריאקט שלנו. אנו צריכים רק לציין שה-id הוא content. את אפליקציית הריאקט אנו יוצרים באמצעות:

## ReactDOM.createRoot

הmethodה זו באה עם האובייקט הגלובלי ReactDOM. האובייקט הגלובלי זהה בא בזכות קובץ ה-`react-dom.js`, שדגנו שיווה בסביבת הפיתוח שלנו. methodה מקבלת ארגומנט אחד שהוא האלמנט שאנו בוחרים כדי להריץ בו את ה-`JSX` של האפליקציה ומחזירה אלמנט ריאקטי שמייצג את האפליקציה. בו יש את methodת `render` שמקבלת את ה-`JSX`.

הציבו את הקוד הזה בין תגיוט הסקריפט שלכם:

```
<script type="text/babel">
  const rootElement = document.getElementById('content');
  const root = ReactDOM.createRoot(rootElement);
  root.render(
    <h1>Hello World!</h1>
  );
</script>
```

שמרו וצפו בתוצאה. אתם תראו Hello World בדף אם הכל תקין. אם לא הכל תקין ואתם לא רואים Hello World הפסיקו לקרוא, פתחו את הקונסולה, צפו בשגיאות וחפשו אותן בגוגל כדי לתקן, ודאו שהעתקתם את הקוד כשורה והמשיכו רק כאשר אתם יודעים שסביבת הקוד שלכם עובדת.

הבה נעבור על הארגומנטים השונים של ReactDOM. נתחל מהשני, ה-target. אני יוצר רפנס לאלמנט שבו אני רוצה להציב את התוכן שלי באמצעות:

```
document.getElementById('content');
```

אני מעביר אותו לקבוע `rootElement`. אתם אמורים להכיר את זה אם יש לכם ידע בג'אווסקריפט. אם לא, אני חזרו על החומר של ג'אווסקריפט ו-HTML (שנמצא גם בספר שלי "לימוד ג'אווסקריפט בעברית"). את הקבוע `rootElement` אני מעביר כארגומנט אל הפונקציה `ReactDOM.createRoot` של DOM שמחזירה לו אלמנט שנכנס אל הקבוע `root`. מעכשו בעצם `root` הוא אפליקציית הריאקט שלי ואני יכול לקרוא למתחות `render` ולהכניס לתוכה ארגומנט של JSX.

עכשו נדבר על הארגומנט הזה, שמורכב שיש בו כרגע רק את המידע הזה:

```
<h1>Hello world!</h1>
```

הוא לא מורכב מדי, בסך הכל תיגת HTML. אבל שמו לב למשהו מעניין – ה-HTML הזה נמצא בתוך קוד ג'אווסקריפט! איך זה יכול להיות? הרי אם תשימוש בקוד HTML ללא מירכאות בקוד ג'אווסקריפט, הקוד ידפיס שגיאה ויפסיק לפעול. ג'אווסקריפט לא מכירה HTML ולא יודעת לעבוד איתנו. איך יכול להיות שאינו משתמש ב-`h1` וב-HTML בג'אווסקריפט ללא מירכאות והקוד לא נופל?

הסיבה היא JSX. זוכרים את המרכיב השלישי בסביבת הפיתוח שלנו, ה-`babel`? הוא אחראי למצוא את כל התגיות של HTML שיש בקוד הג'אווסקריפט ולהמיר אותן למשהו שג'אווסקריפט יודעת להתמודד איתנו. הקוד הזה, שכרגע יש בו HTML פשוט בלבד, הוא JSX – ראשית תיבות של XML JavaScript – והוא אחד מהפיצ'רים החזקים שיש לריאקט. אנו נלמד עליו לעומק בהמשך ונראה איך מהמצב שיש לנו אפליקציית ריאקט אנו מגיעים למצב שבו אנו מפתחים קומפוננטות.

JSX הוא ג'אווסקריפט לכל דבר ועניין. אני יכול לשים אותו במשתנה לצורך העניין:

```
const rootElement = document.getElementById('content');
const root = ReactDOM.createRoot(rootElement);
const value = <h1>Hello World!</h1>;
root.render(value);
```

אפשר גם לשים אותו, כפי שנראה בהמשך, בלולאות ובמקומות אחרים.

**תרגיל**

צרו HTML שמכיל אפליקציית ריאקט במחשב המקומי שלכם. אפליקציית הריאקט תהיה ב-div .my-react-app שייקרא

**פתרון**

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
  <script crossorigin
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
</head>
<body>
  <div id="my-react-app"></div>
</body>
<script type="text/babel">
  const rootElement = document.getElementById('my-react-app');
  const root = ReactDOM.createRoot(rootElement);
  root.render(
    <h1>Hello World!</h1>
  );
</script>
</html>
```

ראשית ניצור קובץ HTML פשוט שבו יש קריאה לשולשת הקבצים שאנו צריכים לאפליקציה ריאקט וועליהם הסבכנו בתחילת הפרק.

הצעד הנוסף הוא ליצור דף שיש לו זו מסויים שאנו רוצים להכניס אליו את אפליקציית הריאקט. במקרה זהה:

```
<div id="my-react-app"></div>
```

השלב הבא הוא ליצור את אפליקציית הריאקט באמצעות:

## ReactDOM.createRoot

הmethod זה מקבל ארגומנט שהוא האלמנט שהאפליקציה תהייה בו. אני מקבל אותו באמצעות `getElementById`, שהוא מתודת ג'אווסקריפט שנמצאת בדף באופן טבעי (בלי קשר לריאקט), ועביר אותו באמצעות משתנה:

```
const rootElement = document.getElementById('my-react-app');
const root = ReactDOM.createRoot(rootElement);
```

הmethod מחזירה את ה-`root` של האפליקציה. אובייקט מיוחד שיש בו מתודות של אפליקציה ריאקט.

כשיש לי את ה-`root` של האפליקציה, אני יכול להריץ בו קוד JSX. אני יכול לעשות כן עם מתודת `render` שמקבלת ארגומנט אחד: JSX פשוט שאינו שונה מ-HTML, מלבד העובדה הפשוטה שהוא בתוך קובץ ג'אווסקריפט:

```
root.render(
  <h1>Hello World!</h1>
);
```

שמירת ה-HTML ופתחה שלו באמצעות הדף תציג לי `Hello World!`. או כל טקסט אחר.

פרק 2

# **בנייה קומפוננטת פונhaziיה ובסביבת העבודה הבסיסית**



# בנייה קומפוננטת פונקציית בסביבה

## העובדת הבסיסית

אחרי שלמדנו לבנות את האפליקציה ואיפילו למדנו על JSX בסיסי, הגיע הזמן למדוד על המבנה הבסיסי של ריאקט – קומפוננטות. אחת מההפקות הגדולות בפיתוח צד לקוח שריект הביאה היא הקומפוננטות. מדובר ביחידות תוכנה קטנות שאפשר להשתמש בהן שוב ושוב במקריםות שונות באתר שלנו. המתכנתים בריאקט יוצרים למשל קומפוננטה אחת של טבלה הניתנת למשון ויכולים להשתמש בה בכל מקום באפליקציה או באתר שלהם. אם הם צריכים לשנות את הטבלה, הם משנים קומפוננטה אחת בלבד. בכל אפליקציה ריאקט יכולות להיות אינספור קומפוננטות. למתכנתים חדשים קל לחשב על קומפוננטות בתור פונקציות.

בתרגיל בפרק הקודם יצרנו אפליקציה ריאקט שבה כתוב "Hello World" או כל טקסט אחר. אם אני רוצה לנתח כמה פעמים "I am learning React!" או "I am learning React!", אני יכול לשכפל את ה-h2 ב-JSX הבא:

```
const rootElement = document.getElementById('my-react-app');
const root = ReactDOM.createRoot(rootElement);
root.render(
  <div>
    <h2>I am learning React!</h2>
    <h2>I am learning React!</h2>
    <h2>I am learning React!</h2>
  </div>
);
```

שימוש לב שאני צריך לעטוף את השכפולים שלי ב-div אחד מكيف, כיוון שפונקציית render דורשת ממוני אלמנט אב אחד. בתוך אלמנט האב אני יכול לשכפל כמה פעמים שאני רוצה את מה שבא לי, אבל זו לא הדרך הריאקטית. הדרך הריאקטית היא ליצור קומפוננטה.

הבה ניצור קומפוננטה שהיא זו שתדפיס עבורנו את:

`<h2>I am learning React!</h2>`

יש שני סוגי קומפוננטות – סוג אחד הוא קומפוננטות מבוססות קלאס, שעליהן למד בהמשך הספר. הסוג השני והנפוץ יותר הוא קומפוננטה של פונקציה, והוא פשוטה למדי. מגדירים אותה באמצעות פונקציה (זו לא הפטעה גדולה, נכון?). כרגע זה די פשוט. הקומפוננטה נראה כך:

```
function Greeting() {
  return <h2>I am learning React!</h2>;
}
```

כדי לשים לב לכמה דברים – ראשית, שם הפונקציה מתייחל באות גדולה. זו קונבנצייה של ריאקט שמחיבת כל קומפוננטה שהיא. שנית, הפונקציה מחזירה JSX. במקרה הזה מדובר ב-HTML פשוט. זה מאפשר לנו להבדיל בקלות בין פונקציה רגילה לבין פונקציה שייצרת קומפוננטה. איך משתמש בקומפוננטה זו? בדיק כמוה HTML. כך:

`<Greeting />`

כלמנט שסגור את עצמו. או כך:

`<Greeting></Greeting>`

אך על פי שהקונבנצייה החד-משמעות היא לכתוב אלמנט שסגור את עצמו.

ואיך הקוד המלא שלנו ייראה כה:

```
function Greeting() {
  return <h2>I am learning React!</h2>;
}

const rootElement = document.getElementById('my-react-app');
const root = ReactDOM.createRoot(rootElement);
root.render(
  <div>
    <Greeting />
  </div>
);
```

אני יכול לשכפל את הקומפוננטה כרצוני, כמובן. הקוד המלא יראה כה:

```
function Greeting() {
  return <h2>I am learning React!</h2>;
}

const rootElement = document.getElementById('my-react-app');
const root = ReactDOM.createRoot(rootElement);
root.render(
  <div>
    <Greeting />
    <Greeting />
    <Greeting />
    <Greeting />
    <Greeting />
    <Greeting />
  </div>
);
```

היתרון הגדול? אם אני רוצה לשנות את הכתוב בקומפוננטה או את האלמנט או להוסיף לקומפוננטה, אני יכול לעשות את זה במקום אחד בלבד, ובבת אחת השינוי הזה ישפייע על כל שימוש ושימוש בקומפוננטה בתוך האפליקציה שלי.

בתוך הקומפוננטה אני יכול להשתמש בעוד קומפוננטות, וזה חשוב מאד. הנה ניצור קומפוננטה נוספת שבה משתמש בתוך קומפוננטת Greeting. משהו בסגנון זהה:

```
function Hello() {
  return <span>Hello,</span>
}

function Hello() {
  return <span>Hello,</span>
}

function Greeting() {
  return <h2><Hello />I am learning React!</h2>;
}

const rootElement = document.getElementById('my-react-app');
const root = ReactDOM.createRoot(rootElement);
root.render(
  <div>
    <Greeting />
  </div>
);
```

אולי הקוד הזה נראה לכם מסובך, אבל הוא ממש לא! ראשית, יש לנו קומפוננטה שמחזירה לנו Hello. היא נראית כך:

```
function Hello() {
  return <span>Hello,</span>
}
```

ניתן להשתמש בקומפוננטה זו בכל מקום – היפשר באפליקציה או בתוך כל קומפוננטה אחרת. במקרה זהה מי משתמש בה הוא קומפוננטת Greeting. איך היא משתמשת בה? בדוק כמו ב-HTML. שם הקומפוננטה כתגיות HTML:

```
function Greeting() {  
  return <h2><Hello />I am learning React!</h2>;  
}
```

אני יכול להשתמש בקומפוננטה זו בכל מקום ב- JSX וafiloo כמה וכמה פעמים, אבל פה הסתפקתי רק בפעם אחת.

אני יכול להשתמש, כמובן, בקומפוננטת Greeting כמה פעמים שניי רוצה. בכל פעם שניי משתמש בה, אני אראה על המסך Hello, I am learning React!. Am I correct? שיפא אם ארצה לשנות את הברכה, יוכל לעשות זאת בקלות דרך שינוי במקום אחד. זה מה שיפא בקומפוננטות ריאקטיות – אפשר לבצע בהן שימוש חוזר (המונח המקביל הוא reuse) בכל מקום. קומפוננטות שמכילות עוד קומפוננטות ובתוכן יש עוד קומפוננטות וכן הלאה; הכל מתנקז בסופו של דבר לאפליקציית ריאקט אחת שמנילה בתוכה אינספור קומפוננטות.

**תרגום:**

צרו אפליקציה ריאקט שבתוכה יש קומפוננטה אחת שנקראת Root. בקומפוננטת Root יש שתי קומפוננטות נוספות, אחת שנקראת soigo ומחזירה JSX שהוא Hello, Prepare to die!, והשנייה שנקראת Greeting ומחזירה JSX שהוא my name is Inigo Montoya. בהפעלת האפליקציה אני אראה שכתוב: Hello, my name is Inigo Montoya, Prepare to die!

**פתרון:**

```
function Inigo() {
  return <span>Hello, my name is Inigo Montoya</span>
}

function Greeting() {
  return <span>prepare to die!</span>
}

function Root() {
  return <span><Inigo />, <Greeting /></span>
}

const rootElement = document.getElementById('my-react-app');
const root = ReactDOM.createRoot(rootElement);
root.render(
  <div>
    <Root />
  </div>
);
```

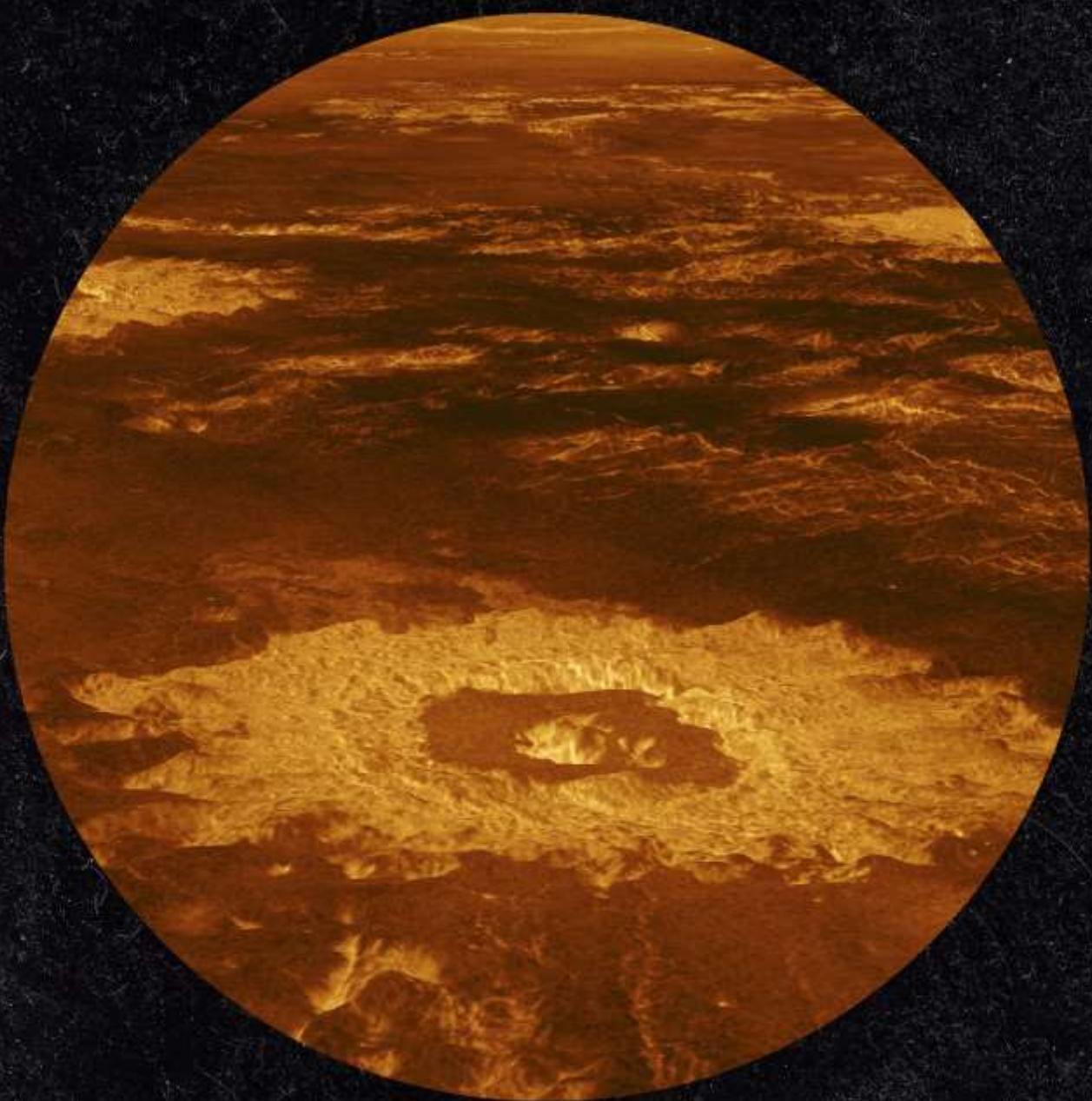
יצרתי שלוש קומפוננטות פשוטות. הראשונה והשנייה soigo ו-Greeting מחזירות JSX פשוט. כדי לשים לב שב- JSX אני חייב לשים אלמנט אב אחד, במרקחה זהה בחרתי ב- span. הקומפוננטות הללו הן קומפוננטות מסווג פונקציה. כדי לשים לב ששמן חייב להתחיל באות גדולות. מהרגע שהגדרתי אותן, אני יכול להשתמש בהן בכל קומפוננטה אחרת בדיקן כמו כל אלמנט HTML אחר.

הקומפוננטה השלישית היא קומפוננטת Root והוא מכילה את שתי הקומפוננטות soigo ו-Greeting. גם פה, חשוב לציין, יש אלמנט אב אחד. גם הוא span.

כל מה שנותר לי לעשות הוא להציב את הקומפוננטה השלישית, Root, באפליקציה שלי.

פרק 3

# בנייה של סביבת ריאקט אמיתית ומורכבת יותר



# בנייה של סביבת ריאקט אמיתית ומורכבות יותר באמצעות Vite

ב פרקים הקודמים למדנו להקים סביבה פשוטה וריאינו איך יוצרים אתרים או אפליקציות ווב באמצעות אפליקציות ריאקט וקומפוננטות פשוטות. סביבת העבודה הזו היא פשוטה מאוד ומלבדת אותנו שリアקט זה לא קסם ולא וודו – אבל רובם המוחץ של המתכנתים לא משתמשים בסביבת ריאקט כזו. היא פשוטה ופרימיטיבית מדי. כמה נוח היה אילו היו בסביבה שלנו שירות מובנה, במקומם לטעון את הקובץ דרך מערכת הקבצים המקומיות, או `reload`, `hot`, שמרפרש אוטומטית את האפליקציה בסביבת הפיתוח בכל פעם שאנחנו עושים שינוי, או דיבאגר מובנה... כל אילו ועוד קיימים ממש מן הקופסה בריאקט.

ישנם כמה פרויקטים המאפשרים בנייה של סביבה טובה עם ריאקט וטייפסקרייפט. נcone לכתיבה של רשות אלו, השניים המובילים הם `Vite` ו-`js-Next`. במספר זה אנו לומדים על `Vite` (הוגים אותה כך: ויט והפירוש הוא מהר בצרפתית). `Vite` זו בעצם סביבת פיתוח מלאה הכוללת שרת שאפשר להפעיל בקלות מכל מחשב. סביבת הפיתוח הזו מבוססת על `js-Node`, אך לא נדרש ידע ב-`Node.js` על מנת לעבוד בה.

בנוסף, ניתן להשתמש בטיפסקרייפט בקלות ב-`Vite` ולכתוב בה והאפליקציה תדאג להמיר את טיפסקרייפט לג'אווהסקייפ特 יחד עם ה- `JSX`. זה מאפשר לי להשתמש בטיפים ומאד מקובל.

אנו לא יכולים להישאר בעולם `Hello World`, ולימוד בניית סביבה אמיתית עם `Vite` הוא חלק בלתי נפרד מהלימוד של ריאקט. הפרק הזה הוא אחד הפרקים הכי חשובים והכי מורכבים שיש בספר וחשוב מאוד לא להירגע ולא להיבהל. אם יש תקלות, כדאי מאוד לפתור אותן. כאמור, העולם של ריאקט הוא עשיר מאוד ומישהו אחר כבר חוווה כל תקלה שתחוור בדרך (אם תחווו). חיפוי מהיר של התקלה בגוגל יסייע לכם מאוד.

## התקנת `Node.js` על המחשב שלכם

ראשית, אתם חייבים להתקין את `Node.js` על המחשב המקומי שלכם שעובדים בו. `Node.js` הוא סביבה המאפשרת להריץ ג'אווהסקייפט על גבי מערכת ההפעלה, במקרה זהה מערכת ההפעלה

שלכם. ג'אווהסקריפט, למי שלא יודע, היא שפה גמישה מאוד ואפשר להפעיל אותה לא רק בסביבת הדפדפן, כמו שאנחנו עושים בריект, אלא גם בסביבת מערכת הפעלה/שרתיים – ואת זה עושים באמצעות `Node.js`. הפלטפורמה זו ניתנת להתקינה בכל מערכת הפעלה שהיא ובקלות. בחרו את מערכת הפעלה שלכם והתקינו את `Node.js` לפי ההוראות.

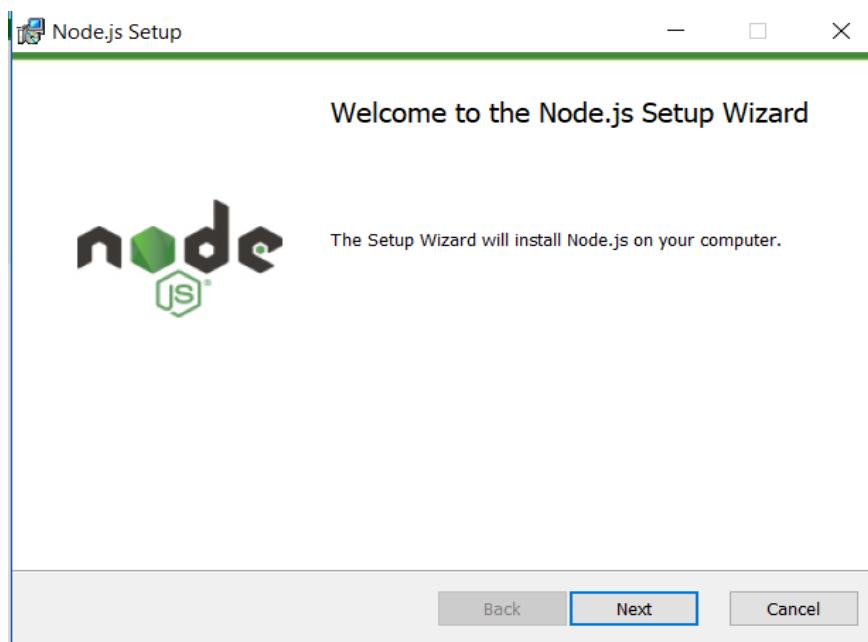
## התקנה על חלונות

ההתקנה של `Node.js` על חלונות היא פשוטה. נקליד בגוגל `Node.js Download` או ניכנס אל:

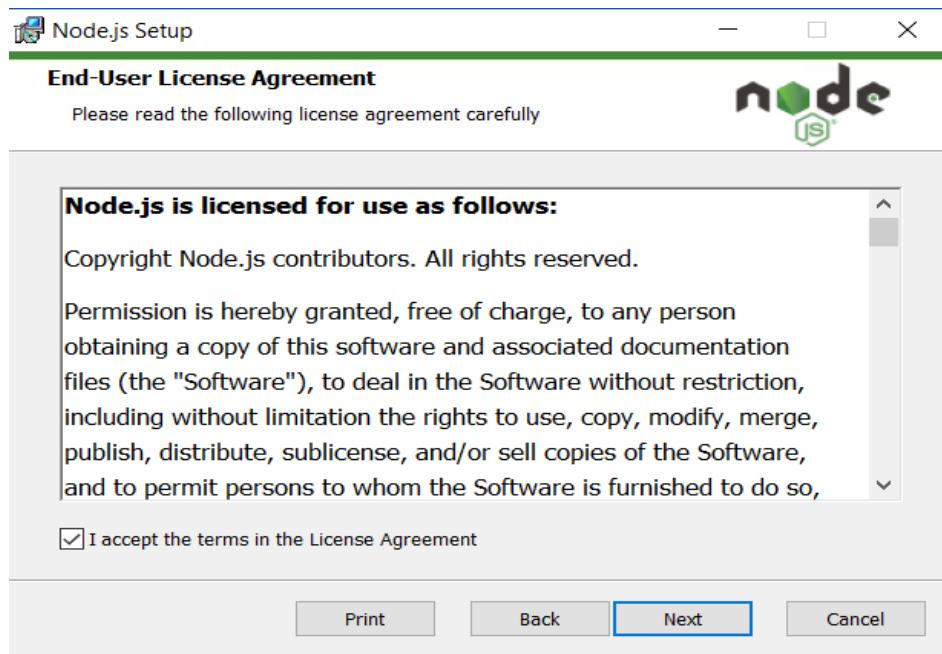
<https://nodejs.org/en/download/>

ano נבחר בגרסת LTS – ראש תיבות של "גרסה לטוח אורך", ונבחר במערכת הפעלה שלנו –

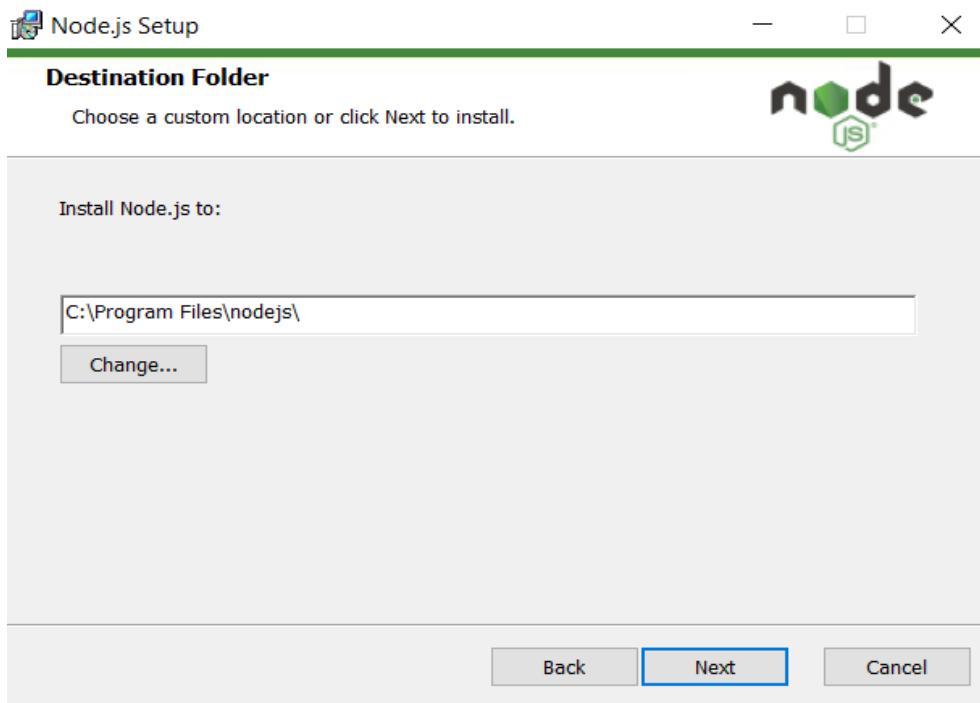
אם מדובר בחלונות, יש לנו `installer` נוח. מוריידים, לוחצים על התוכנה שיורדת:



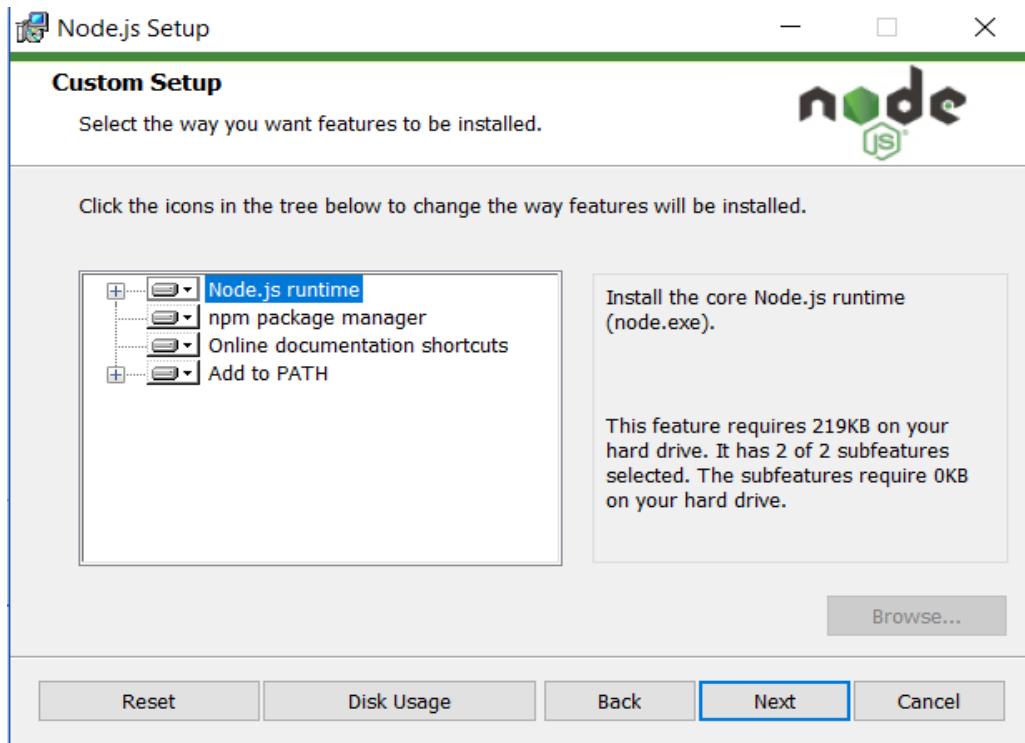
**מקבלים את התנאים:**



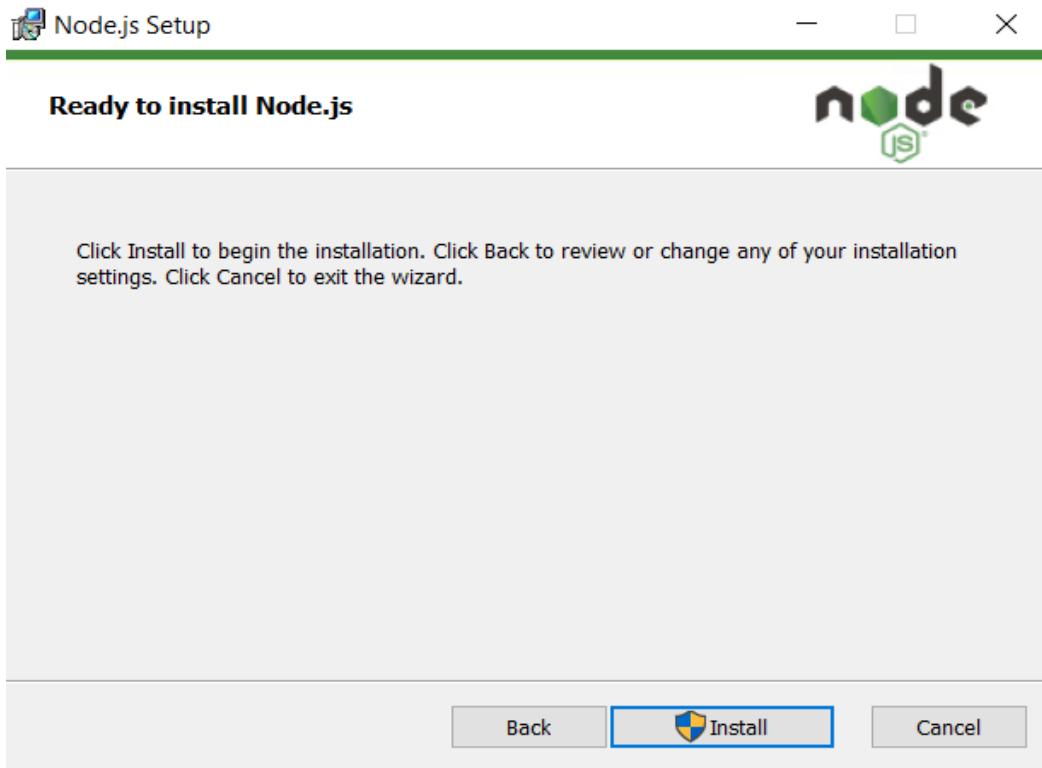
:Next לוחצים על



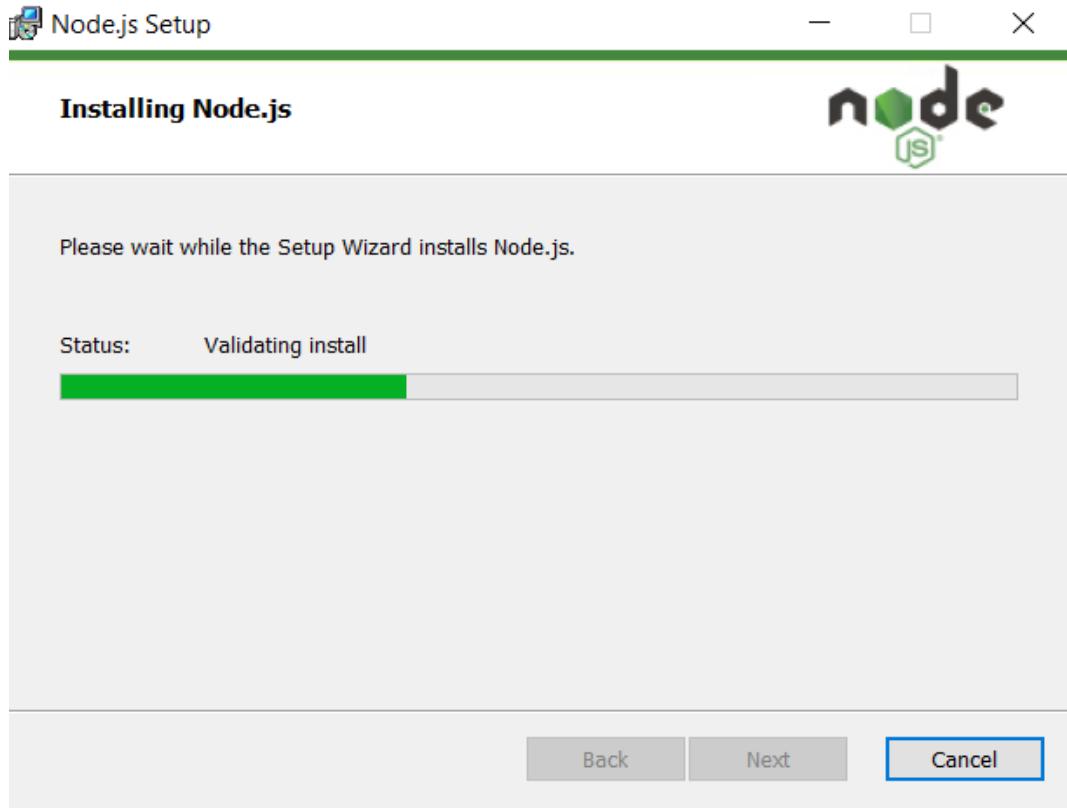
ושוב על :Next



לחיצה על Install לבסוף מתקין את התוכנה:



כל מה שנותר הוא לחכות לסוף ההתקנה:



## התקנה על מק

ההתקנה של Node.js על מק פשוטה מאוד. נקליד בグוגל Node.js Download או ניכנס אל:

<https://nodejs.org/en/download/>

אנו נבחר בגרסה LTS – ראשיתיבות של "גרסה לטווח ארוך", ונבחר במק – ייד קובץ rpm שהוא אפשר להתקין כמו כל תוכנה אחרת בהנחה שהמחשב שלכם הוא לא מחשב ארגוני שמנוע התקנות מהאינטרנט. ההתקנה היא פשוטה ביותר.

אם אתם משתמשים ב-Zsh או ב-Bash אני ממליץ להתקין את Node.js בעזרת homebrew באמצעות הפקודה brew install node (אם homebrew מותקנת אצלכם, וכדאו שהוא מותקן). כך או אחרת, לאחר ההתקנה, כניסה לטרמינל והקלדה של `v -v` יראו לכם את מספר הגרסה.

## התקנה על לינוקס

אם אתם משתמשים בדביאן, בדרך כלל, ברוב ההפצות, sudo apt-get install node יטפל בהתקנה, אך ייתכן שתתקינו גרסה ישנה של js.Node, זהה עלול להיות בעייתו. למרות הפיתוי, קראו לפני ההתקנה את המדריך המלא לכל ההפצאות של לינוקס, שסביר על ההתקנות: <https://nodejs.org/en/download/package-manager/>

אני יוצא מנקודת הנחה שמשתמשים בלינוקס הם מיומנים בהרבה משתמשי חלונות ויודעים להתקין חבילת תוכנה ללא הסברים נוספים. נך או אחרת – לאחר ההתקנה, כניסה לטרמינל והקלדה של node יראו לכם את מספר הגרסה בדיקם כמו במק.

## עבודה עם טרמינל

כמו כל אפליקציית js.Node, גם Vite מצריכה אותנו לעבוד עם טרמינל. טרמינל הוא הממשק שבו אני מקליד פקודות למערכת הפעלה. כל מתכנת עובד עם טרמינל כאשר הוא מתחבר מרוחק לשרתים שונים וזה ידע שימושי למדוי. אנו נלמד בעת איך לתחבר לטרמינל ואיך לעבוד אליו בחלונות. אני לא מסביר על טרמינל במק או בלינוקס כי אני יוצא מנקודת הנחהשמי שיש לו את מערכות הפעלה האלו יודע איך להשתמש באופן בסיסי בטרמינל.

## הפעלת הטרמינל

בחלונות הגישה לטרמינל פשוטה למדוי. בחלונות 10 ו-11 לוחצים על הזוכנית המגדלת, מקלידיים cmd או לוחצים על אנטר.



מיד מקבלים מסך שחור. לוותיקים בינוינו הוא יזכיר את מסך DOS הישן של לפני 20 שנה.

```
Command Prompt
Microsoft Windows [Version 10.0.17134.950]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\barzik>
```

זה הטרמינל של חלונות. לפני כמה שנים, כך נראה מחשבים. זה המחשב שלכם, אבל המשק הוא לא גרפי אלא טקסטואלי. יכול להיות זהה יראה לכם מישן ולא רלוונטי, בטח ובטע לעומת הממשקים האחרים שאתם מכירים, אבל כנה רוב המתקנים עובדים – מול הטרמינל של חלונות, של מק או של לינוקס. חשוב מאוד להבין איך עובדים איתנו.

כשתיכנסו לטרמינל, מצד שמאל תוכלו לראות את המיקום שלכם. אצל המיקום הוא:  
C:\Users\barzik

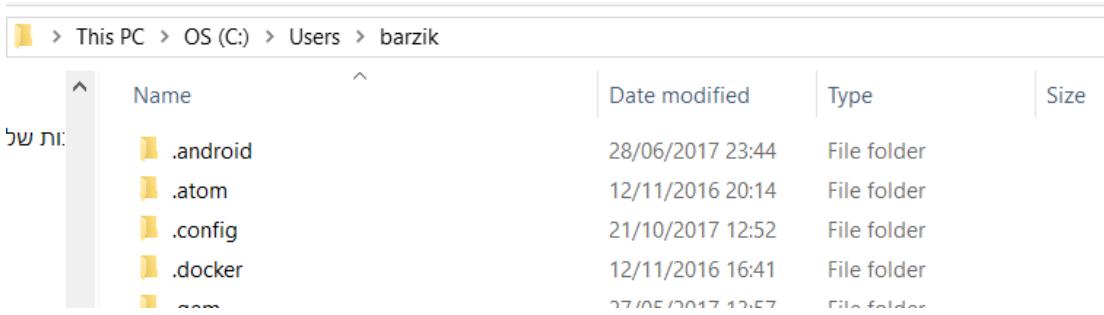
אצלכם המיקום יהיה שונה בהתאם לשם המשתמש שלכם. הבה ננסה להקליד פקודה. הקלידו dir והקישו על אנטר. אনטר בטרמינל הוא "שידור". אם תעשו את זה כמו שצرين, תראו מהו זהה:

```
C:\Users\barzik>dir
Volume in drive C is OS
Volume Serial Number is 0C36-DF83

Directory of C:\Users\barzik

08/17/2019  11:07 AM    <DIR>      .
08/17/2019  11:07 AM    <DIR>      ..
06/28/2017  11:44 PM    <DIR>      .android
11/12/2016  09:14 PM    <DIR>      .atom
07/28/2019  07:30 PM            3,842 .bash_history
10/21/2017  12:52 PM    <DIR>      .config
.
```

זאת בעצם כל רשימת הקבצים בתיקייה שלכם. אם תפתחו את סיר הקבצים המובנה בחלונות ותחפשו את התיקייה, תראו שהיא שהפקודה dir נותרת זהה לתצוגה שלכם.



	Name	Date modified	Type	Size
זהות של	.android	28/06/2017 23:44	File folder	
	.atom	12/11/2016 20:14	File folder	
	.config	21/10/2017 12:52	File folder	
	.docker	12/11/2016 16:41	File folder	
	.vscode	7/5/2017 23:00 23/05/2017 23:00	File folder	

## ניווט בטרמינל

אתם יכולים לנועט בטרמינל ולהגיע לתיקיות אחרות. למשל, אם יש תיקיית Documents במיקום שלכם, אפשר להגיע אליה. נסו להקליד למשל: cd Documents.

תגיעו לתיקיית Documents שיש תחת השם שלכם. אם תקלידו dir ותצפו בתוכן התיקייה, תראו שהיא זהה לתיקיית My Documents מסיר הקבצים.

אפשר "לעלות" לתיקייה אחת למעלה באמצעות .. cd. נסו לעלות שוב ושוב עד שתגיעו לתיקייה הראשית, הלווא היא ..

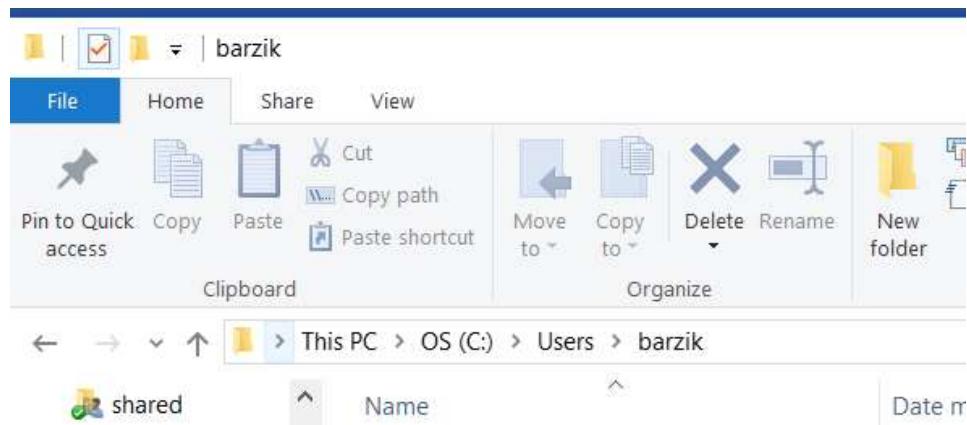
אנו יכולים לנועט שוב בחזרה באמצעות cd ושם התיקייה. אפשר גם להקליד ישירות את הנתיב:

```
C:\Users\barzik\Documents>cd ..
C:\Users\barzik>cd ..
C:\Users>cd ..
C:\>cd Users\barzik\Documents
C:\Users\barzik\Documents>
```

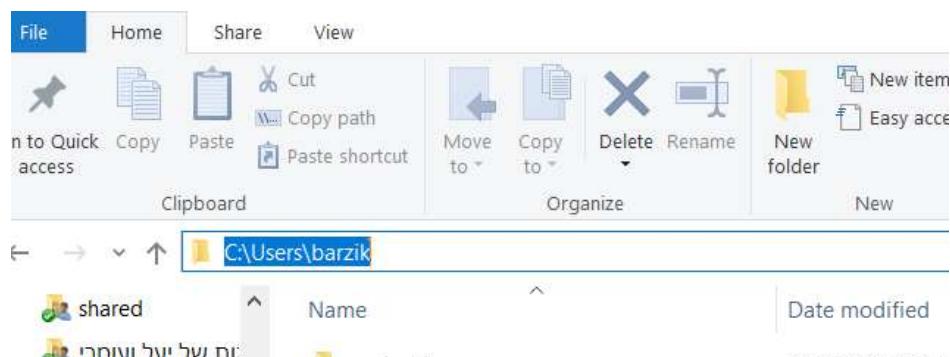
טיפ קטן – אפשר להקליד tab כדי לבצע השלמה.

## מציאת מקומות בטרמינל דרך חלונות

אם אתם רוצים להגיע לתיקיה מסוימת ולא בטוחים מה מיקומה, מצאו אותה בסיר הקבצים הגרפי ולהצלו על הוכתרת. למשל, התיקייה הזו:



אם אניalach על המיקום, אקבל את מיקום של התיקייה בנתיב מסודר שבו אני יכול להשתמש בטרמינל:



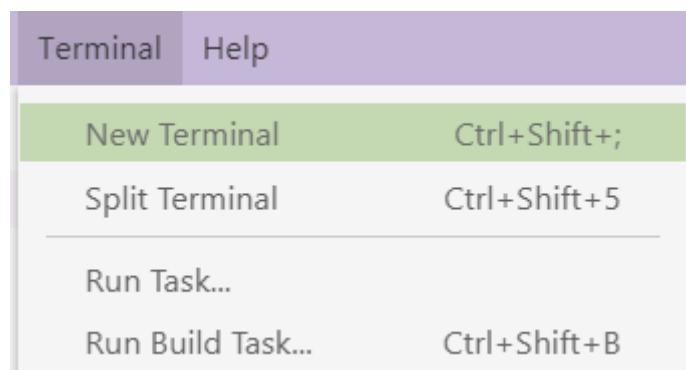
אני יכול להעתיק את הנתיב אל הטרמינל. להקליד:  
`cd C:\Users\barzik`

ואז ללחוץ על אנטר ולהגיע אל היעד המבוקש.

## טרמינל ב-Visual Studio Code

אפשר לעבוד עם הטרמינל דרך ה-IDE החינמי Visual Studio Code, שומומלץ לכל מפתח ג'אווסקריפט. העורך הזה, מבית מיקרוסופט, הוא חינמי, לא מכבד על המחשב וגם קל להסורה. אם אתם לא משתמשים בו, אני ממליץ בחום שתתקינו אותו באמצעות כניסה לאתר הרשמי שלו:  
<https://code.visualstudio.com/>

אם הוא מותקן אצלכם, פתחו את תיקיית הפרויקט שלכם, לחצו על לשונית הטרמינל לעלה ובחורו ב-`New Terminal`:



ב-IDE שלכם יופיעחלון של טרמינל שנפתח כבר במקום הפרויקט שלכם. אפשר גם ליצור תיקיות בטרמינל, למחוק תיקיות, לשנות ולעורך קבצים ולעשות פעולות נוספות. רבים מהמתכנתים עובדים דרך הטרמינל של Visual Studio Code.

## בדיקות גרסת `js.Node` דרך הטרמינל

הפעולה המרכזית שאנו נעשה בטרמינל היא בדיקה שהתקנת `js.Node` שעשינו תקינה. אנו נקליד:

```
node -v
```

אם קיבל שגיאה, סימן שההתקנה לא הייתה תקינה. נסו להפעיל מחדש את המחשב או להתקין מחדש את `js.Node`.

אם הכל תקין, תראו את הגרסה של Node.js שהותקנה אצלכם:

```
C:\Users\barzik\Documents>node -v
v10.15.3
```

אם התקנתם את Node.js וכשאתם מקלידים `-v` בטרמינל אתם מקבלים את הגרסה, אפשר להתקדם לשלב הבא – עבודה עם Vite.

## עבודה עם Vite

פתחו את הטרמינל ונווטו באמצעות `cd` לתיקייה שאתם רוצים שהפרויקט שלכם יהיה בה. אצלי למשל כל האפליקציות נמצאות בתיקיית `local`. יצרתי תיקייה `local` תחת המשמש שלי ונכנסתי אליה באמצעות:

```
cd C:\Users\barzik\local
```

לחופין, פתחו את Visual Studio Code, צרו באמצעותו פרויקט במקום מסויים ובאמצעות לחיצה על לשונית `New Terminal` יפתח לכם בתחום התוכנה חלון טרמינל שנמצא במקום של הפרויקט שלכם.

כשאני נמצא בתיקייה שבה אני רוצה להקים את פרויקט הריאקט הראשון שלי, אני מקליד בטרמינל:

```
npm create vite@5.2.3 my-app -- --template react-ts
```

`npm create` הוא פקודה הפעלה של Node.js.  
`vite` הוא שם התוכנה שאנו מפעילים וגרסתה – 5.2.3. ניתן להשתמש גם ב `latest`.  
`my-app` הוא שם האפליקציה שלנו. כשנפתח תוכנה אמיתית אנו נקרא לה מן הסתם בשם משמעותי יותר. כרגע נבחר ב-`my-app` – האפליקציה שלי.  
`--template react-ts` – הוא הוראה ליצור את הפרויקט עם ריאקט וטייפסקרייפט מובנה.

אם יש לכם `js.js` במערכת והכל תקין, ה התקינה תעבור בקלות. הוא נמשכת כמה דקות טובות, אין מה לחושש. במהלך הדקות האלו התוכנה מורידה את כל המרכיבים מהרשת על מנת ליצור אתכם במחשב סביבת עבודה מלאה של ריאקט.

**Done. Now run:**

```
cd my-app
npm install
npm run dev
```

זה הפלט שאמורים לראות. בסיום ההתקנה אתם אמורים להיות מסוגלים להקליד שוב בטרמינל. נוצרה לכם תיקייה שנקראת `app-my`. היכנסו אליה באמצעות:

```
cd my-app
```

התקינו את המודולים של המערכת באמצעות

```
npm install
```

והפעילו את המערכת החדשה שלכם באמצעות:

```
npm run dev
```

מדובר בפקודה של `js.js` שפשות מפעילה סкриיפט שנקרא `dev`. אם הכל תקין, הדבר הראשון שתשים לב אליו הוא שמודפס פלט בטרמינל של סיום פעולה:

```
VITE v5.2.8 ready in 723 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

אם תכנסו אל הכתובת המצוינת שם שהוא `http://localhost:5173` תראו שנפתח לכם חלון של דף פון שמכיל את האפליקציה שלכם! זה לא קסם. מה שתהליך `js.js` עושה הוא לייצר שרת על המחשב שלכם אחורי הקളעים ולטען את רכיבי הריאקט. אם תסתכלו על כתובות

הדף תוכלו לראות שמדובר בכתובת של localhost: 5173. הכתובת זו מוחלקת לשני חלקים. localhost הוא בעצם הכתובת של המחשב שלכם ו-5173 הוא הפורט (נתב הגישה). כיוון שהאתר הוא פנימי, אנו לא צריכים ליצור לו דומיין. בשלב מאוחר יותר נלמד איך להעלות את האתר שלנו מהשרת הפנימי אל שרת חיצוני. כרגע זו פשוט סביבה לימוד ופיתוח מצוינה.

אם תפתחו את Visual Studio Code (או כל עורך טקסט אחר) תוכלו לראות שכבר יש מבנה בסיסי לאפליקציה. בתיקיה הראשית אנו נראה שקובץ ה-HTML קיים והוא כבר מכיל:

```
<div id="root"></div>
```

אפליקציה פשוטה. בתיקיה הראשית גם היא כבר מוגדרת בתיキות src ושם main.tsx וכן כללת הפניה לקומפוננט App פשוטה:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.tsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

איך מתבצעת ההפניה? באמצעות import, שנרchieב עליו בהמשך. באיזה קובץ נמצאת App? רואים את זה ב-import. בקובץ App.tsx באוטה תיקיה:

```
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

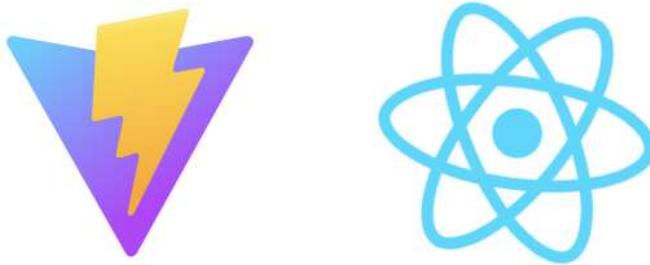
  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React
logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.tsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

אם נשנה משהו בקומפוננטה, למשל נוסף שורה כמו:

```
<p>
  שלום עולם
</p>
```

ונשמר, יוכל לראות שבמיטה קסם, גם האפליקציה שלנו השתנתה ומופיע בה "שלום עולם".



# Vite + React

שלום עולם

זה קורה כי השרת מאמין לכל הקבצים בתיקייה ואם אחד מהם משתנה הוא טוען את עצמו מחדש. נשמע מסובך ופלאי, אם כי זה לא מסובך להבנה וכל מתכנת שמכיר `Node.js` יוכל לייצר זהה דבר. אבל אנחנו לא צריכים להתאמץ – Vite יכולה לעשות את זה עבורנו. ממש נפלא. בסופו של התהליך יש לנו אפליקציה בסיסית שעבדת ואנחנו יכולים ליצור לה קומפוננטה ראשונה.

שמות הסיווגות של קבצים המכילים קומפוננטות של ריאקט הן `jsx` ולא `ts`. זו וריאציה של `JSX` רק עם טיפוס קריפט. בפרויקטים שאין בהם טיפוס קריפט, הסיווגות הן `tsx`. אנו כ毋ון עובדים עם

טיפוסקריפט אז בכל קובץ שיש בו קומפוננטה של ריאקט או קוד ריאקטיבי אנו נשמר אותו בסימת `ts`. שם פורמט הנתונים הוא `XSL`, אך הסימת היא `ts`. אם יש לנו קבצים שיש בהם רק טיפוסקריפט ללא קומפוננטות של ריאקט, הסימת שלהם תהיה `ts` ולא `ts`. אנו נרחב על כך בהמשך, אבל כרגע כלל האצבע הוא סימת `ts` לקומפוננטות של ריאקט, סימת `ts` לקבצים שבהם אין קומפוננטות כלו.

## קשיים ותקלות

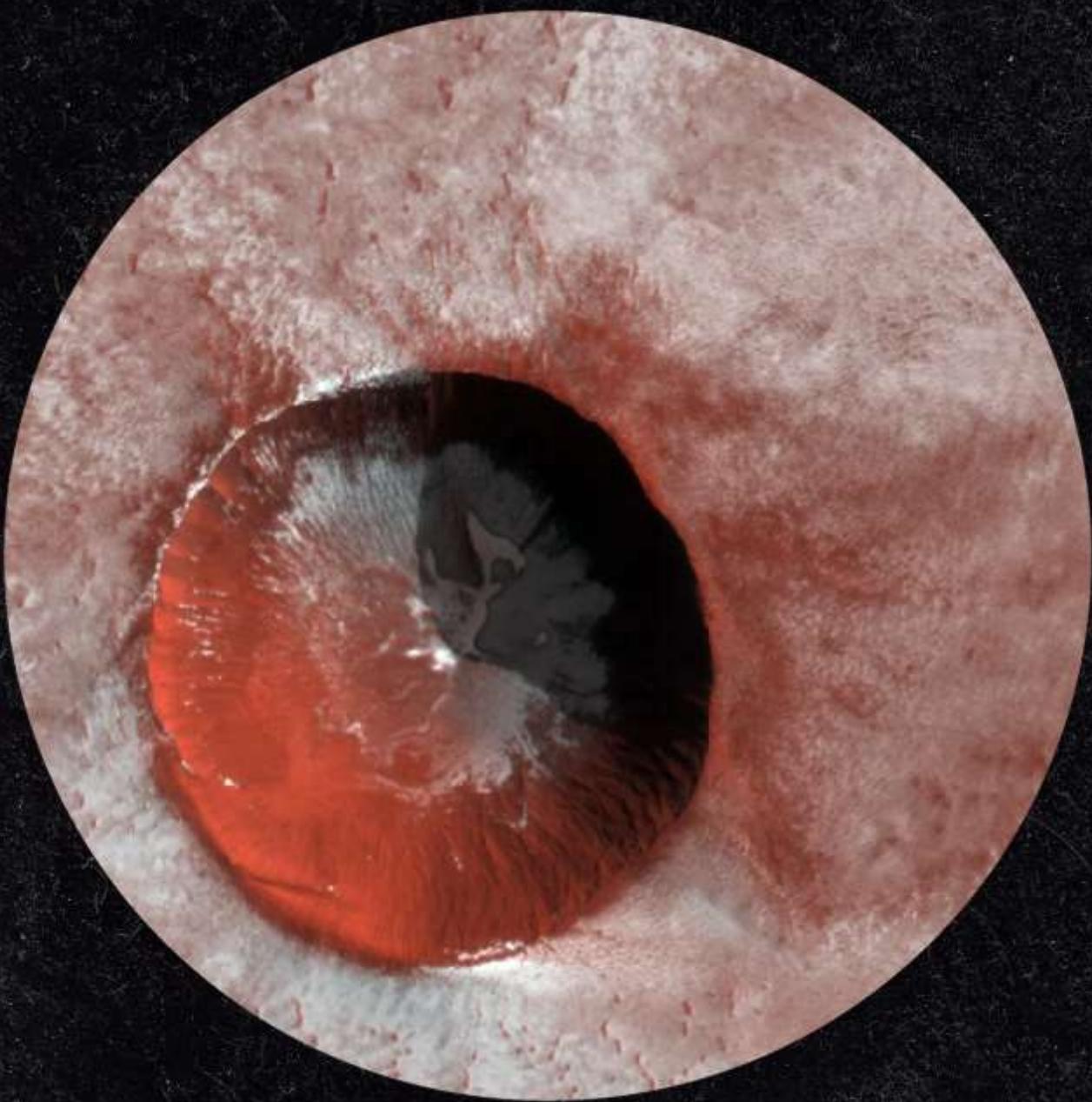
אם נתקלتم בקשאים בתהיליך ההתקנה של `Vite`, אל דאגה! יש לעיתים שנובעות משינויים במערכת הפעלה, אנטי-וורים שפותחים מתעורר או בעיה אחרת שנובעת מכך שמערכות הפעלה שלנו שונות. אבל כדאי מאד לא לוותר. אם יש הودעת שגיאה – פשוט חפשו אותה בגוגל או בידקו עם בינה מלאכותית כמו צ'אט GPT. `Vite` היא כל כך פופולרית, עד שיש תיעוד רב מאוד לגבייה בראשת וגם תיעוד של כל תקלה אחרת. קיבלתם שגיאה אדומה ומפחידה? הדבקו אותה בחלונית השאלה של צ'אט GPT עם השאלה: "How to solve" עם השאלה: "How to solve" שגיאה. חפשו אותה בגוגל, התיעדו לגבייה בקובצת פיסבוק או באתר StackOverflow. אל תדלגו על בניית סביבת העבודה זו כי היא הבסיס העיקרי לעובדה עם ריאקט.

## Create React App

בעבר הייתה סיבה רשאית לפיתוח ריאקט בשם Create React App. היא עבדה בדומה ל-`Vite` ויש אליה לא מעט הפניות ואיזוריים בספרות מקצועית. היא דומה מאד באופן התפעול שלו ל-`Vite`. אך הוצאה לגמלאות ונחשבת מושנת כיום ולא מומלץ להשתמש בה. במהדורות קודמות של הספר, השתמשנו ב-`Create React App` אך הספר עודכן ל-`Vite`.

פרק 4

# בנייה קומפוננטה לאשונה בסביבה מודרנית



# כתיבת קומפוננטה ראשונה ב-Vite

אחרי שבנו את סביבת העבודה, הגיע הזמן לבנות את הקומפוננטה הראשונה. אנחנו כבר לא בסביבת "שלום עולם" ובסביבת לימוד פשוטה אלא בסביבה אמיתית יותר. אנחנו לא נכניס את הקומפוננטה שלנו באותו קובץ של App.ts. כלל אצבע חשוב מאוד: כל קומפוננטה נמצאת בקובץ נפרד משלה ועומדת בראשות עצמה. מקובל מאד שם הקובץ יהיה זהה לשם הקומפוננטה ויתחיל באות גדולה. במערכות גדולות מקובל להציב כל קומפוננטה בתיקייה משלה.

הבה ניצור קומפוננטה בשם Greeting שבה יהיה כתוב "שלום עולם". ראשית – הקובץ: ניצור Greeting.tsx תחת תיקיית src. שמו לב שם הקובץ מתייחס ב-G גדולה. הקומפוננטה זו אינה שונה בכלל מקומפוננטה שלמדנו לעבד אותה בסביבת העבודה הקודמת. היא קומפוננטה פונקציונלית שנראית כך:

```
function Greeting():JSX.Element {
    return <span>שלום עולם</span>
}
```

סוג המידע הבסיסי שקומפוננטה ריאקט מחזירה הוא JSX.Element ואנו נגדי לו.

כיוון שנקרה לקובץ זהה באמצעות import, אנו צריכים ליצא את הפונקציה בדרך מסויימת. אין מייצאים? באמצעות סינטקס שנקרה export default וمجע לפני הפונקציה שמיצאים. אנו נהריב על import/export בהמשך, אבל גם בלי להבין עמוק זה הגיוני – אנחנו שמים את הקומפוננטה בקובץ נפרד – ולקובץ זהה אנו עושים import. אנחנו חייבים לעשות export בצד השני. את ה-export עושים כך:

```
export default function Greeting():JSX.Element {
    return <span>שלום עולם</span>
}
```

הבה נשתמש ב-App.tsx. איך? ב-App.tsx ניבא אותו ונשתמש בו כרגיל, כפי שלמדנו בפרק על קומפוננטה בסביבה הפשוטה.

בקובץ App.tsx נעשה import לkomponeneta שלנו באמצעות:

```
import Greeting from './Greeting';
```

מהרגע שעשינו import, אנחנו יכולים להשתמש ב-Greeting أيiph שאנחנו רוצים. ראשית אפשר להציב אותה ב-App.tsx:

למשל:

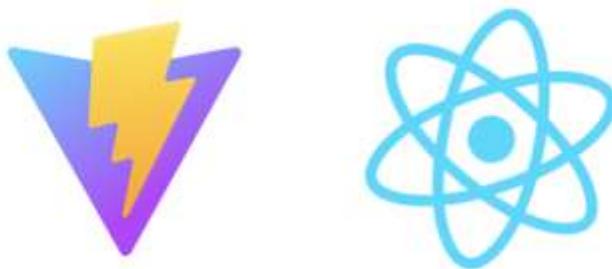
```
function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React
logo" />
        </a>
      </div>
      <Greeting />
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.tsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}
```

או להשתמש בה כמה פעמים. למשל:

```
<Greeting />
<Greeting />
<Greeting />
<Greeting />
```

זה בדיקן כמו הדוגמה פשוטה. ברגע שיש לי קומפוננטה אחת, אני יכול להשתמש בה כמה פעמים. זה שהוא בתוך קובץ ושאנו חנו משתמשים ב-`import\export` מסתורי לא אמור להפריע לנו. בהמשך נלמד עוד על `import\export`. מהרגע שנשמר גם את `Greeting.ts` וגם את `App.ts` – נראה מיד את התוצאה על האפליקציה בלי שהיא צריכה צורך ללחוץ על F5 ולרפרש:



# Vite + React

שלום עולם

שלום עולם

שלום עולם

שלום עולם

הڪפיצה מסביבה לימודית פשוטה לשביבת פיתוח מורכבת נראה קשה וمضיכה, אבל ככל שתתאמנו יותר – כך תתרgalו אליה. יש בה המון דברים טובים – ראשית יש לנו את `babel` שmagiu

בוחנים, יש לנו שרת פיתוח, בדיקות (בהמשך נלמד על הבדיקות) וגם בודק תקינות קוד של eslint שמתכוון על בעיות שימושנו מהאפליקציה עצמה שלנו לרווח. והכל באמת בא מהקופסה. אז גם אם זה נראה מורכב, נא לא להיבהל וلتתרגל על הסביבה הזאת. זו הסיבה שבסוף כל יום תפגשו במקומות העבודה.

## החלפת קוד App.tsx

ב-`App.tsx` יש קוד דוגמה המציג שימוש של יבוא תМОנות וקבצים. על מנת לפשט את הדוגמאות ולהקל על העבודה, אנו נערוך אותו לקובץ פשוט יותר:

```
import './App.css'

function App():JSX.Element {
  return (
    <>
      OUR CODE WILL BE HERE
    </>
  )
}

export default App
```

את כל דוגמאות הקוד והקומפוננטות שניצור מרגע זה, אנו נדגים באמצעות `App.tsx` זהה ונכניס את הקוד שלנו במקום `OUR CODE WILL BE HERE`.

שימוש לב שיש כאן את `<>` ואת `</>` - אנו נסביר עליהם בשלב יותר מאוחר. כרגע יש להתייחס אליהן כנתון.

### הערה בנוגע לריקט מגרסה 16 ומטה:

אם תשתמשו בקוד שלעיל בריקט מגרסה 16 ומטה, תראו שאתם מקבלים שגיאה:

'React' must be in scope when using JSX in `react/react-in-jsx-scope` השגיאה הזאת מرجזת על כך שהוא חסר. מה חסר? במקרה הזה הריאקט עצמו. בעוד בסביבת הלימוד הפשוטה שעלייה למדנו, ריאקט נמצא בכל מקום כי הכל היה בקובץ אחד, בריקט

מגרסה 16 אנחנו חייבים לכלול בכל קומפוננטה וקומפוננטה את כל הרכיבים שאנו צריכים להפעלה. במקרה זהה: ריאקט. אנו חייבים "לייבא" את ריאקט ועושים את זה באמצעות הבא:

```
import React from 'react';

export default function Greeting():JSX.Element {
  return <span>שלום</span>
}
```

از יש לנו כאן ייבוא של ריאקט, שהוא באמצעות פשוט וחינוי בכל קומפוננטה, יש לנו ייצוא, שיהיה חשוב כשרצהה להשתמש בקומפוננטה, ויש לנו את הקומפוננטה שלנו, שבמקרה הזה לא עשו המון אלא רק מחזירה טקסט. והכל נמצא בקובץ בוודד אחד שנקרא `Greeting.tsx`. זה אחד מהשינויים המרכזיים בין ריאקט 16 לריאקט מגרסאות חדשות יותר. אם אתם לומדים מהספר הזה ומשתמשים בגרסה העדכנית של ריאקט, לא תקבלו שגיאה כזו. אך אם אתם עובדים על ריאקט בסביבה אחרת, ישנה יותר, תהיו עירוניים לעניין זה.

**תרגיל:**

באפליקציית Vite צרו קומפוננטה Shenkarat Welcome. הקומפוננטה יושבת ב-  
.src/Welcome.tsx

בקומפוננטת Welcome יש שני קומפוננטות נוספות: הראשונה נקראת So Iigo ומחזירה JSX שהוא Hello, my name is Inigo Montoya והשנייה נקראת Greeting ומחזירה JSX שהוא Prepare to die!. בעמود הראשי של Vite אני אראה שכתוב:

Hello, my name is Inigo Montoya, Prepare to die!

**פתרונות:**

התרגיל זהה לתרגיל של הפרק על סביבת פיתוח פשוטה, אבל הפעם ניצור אותו בסביבת פיתוח מורכבת יותר. הקומפוננטה src/Welcome.tsx קוראת לשתי קומפוננטות נוספות: Inigo.tsx ו-Greeting.tsx. אנו ניצור שלושה קבצים בתיקיית src:

Welcome.tsx

Inigo.tsx

Greeting.tsx

הקומפוננטות Greeting.tsx ו-Inigo.tsx הן קומפוננטות באמת פשוטות. הדבר היחיד שהוא צריך לזכור זה את import React from 'react'. כך הן נראות:

Inigo.tsx

```
export default function Inigo():JSX.Element {
  return <span>Hello, my name is Inigo Montoya</span>
}
```

## Greeting.tsx

```
export default function Greeting():JSX.Element {  
  return <span>prepare to die!</span>  
}
```

איך אני משתמש בהן? מבצע להן import מתוך Welcome.tsx ומשתמש בהן ב-JSX כרגיל. בדוק כדי שהשתמשתי בסביבת הפיתוח הפשוטה. כל מה שאני צריך לזכור הוא לבצע import. זה הכל.  
כל xsx תיראה:

```
import Greeting from './Greeting.tsx';  
import Inigo from './Inigo.tsx';  
  
export default function Welcome():JSX.Element {  
  return <span><Inigo />, <Greeting /></span>  
}
```

שימוש לב שאני מבצע `export default` לקומפוננטה `Welcome`. נותר לי רק לייבא את הקומפוננטה `Welcome` למקום שאני רוצה להשתמש בה, ב-`App.tsx`:

```
import './App.css'
import Welcome from './Welcome.tsx';

function App():JSX.Element {
  return (
    <>
      <Welcome />
    </>
  )
}

export default App
```

השוני בין סביבת העבודה הפשטota יותר לו של Vite הוא ה-`import export`. בעצם כל קומפוננטה עושה `import` לדברים שהוא צריכה ו-`export` לעצמה, כדי שאחרות יוכל להשתמש בה.

פרק 5

# EXPORT / IMPORT



# Import\Export

מדובר בחלק אינהרנטי מג'אווסקריפט (וכמובן בטיפוסקריפט), שחשיבות לכל סקריפט בג'אווסקריפט מורכבת. המנגנון זהה מאפשר לנו לעבוד עם כמה וכמה קבצים ולנהל את הקוד שלנו בקלות. בעבר היה מקובל לכתוב את כל הקוד בקובץ אחד וזה כמובן קשה מאוד לניהול ולבוגה. כפי שראינו בסביבת הפיתוח המודרנית, כדאי לעבוד עם חלוקה של קבצים ועם `import` ו-`export`.

ההיגיון הוא שכל רכיב תוכנה – משתנה, פונקציה, קלאס, קבוע – יכול להיות מיובא על ידי כל רכיב תוכנה אחר. התנאי היחיד שאנו צריכים הוא ליצא באמצעות המילה השמורה `export`. יכולות להיות כמה הוצאות `export` בקובץ אחד. כשיש לנו רכיב תוכנה שמייצא את עצמו – אפשר לייבא אותו.

במקום שיש לייבא חייב להיות ייצוא. הייזוא הבסיסי, הפשוט והקל להבנה מתקיים עם שתי מיללים שמורות: `default` ו-`export`. המשמעות של `export` היא ייצוא והמשמעות של `default` היא ברירת מחדל. כאשרנו כתבים שם של קלאס, פונקציה או משתנה ולפניהם מציבים `export default`, אנו מייצאים את אותו קלאס, פונקציה או משתנה.

הבה נדגים ייזוא פשוט. ניצור קובץ בשם `Vars.js` שיכיל משתנים. ניצור פרויקט של Vite קובץ זה ושם נגדיר משתנה וניצא אותו. בתיקיית `src` ניצור קובץ `Vars.js` וכל מה שהוא בו זה:

```
export default 'Hello';
```

שימוש לב שאין פה קלאס, אין פה משתנים – אני מיצא טקסט בלבד. כדי להשתמש בו, אני רק צריך לעשות `import`. ניכנס ל-`App.tsx`, ניבא אותו ונדייס אותו בקונסולה. הקוד שלנו ייראה כך:

```
import './App.css'
import myVar from './Vars.tsx';

function App():JSX.Element {
  console.log(myVar);
  return (
    <>
      OUR CODE WILL BE HERE
    </>
  )
}

export default App
```

מה שמעניין אותנו הוא ה-`import` שמופיע בשורה השנייה. ה-`import` מביא לתוכה משתנה שאנו מגדיר את כל מה שמופיע אחרי ה-`export default`. במקרה זהה הגדרתי משתנה בשם `myVar`. אני יכול פשוט להדפיס אותו עם `console.log`. אסור לנו לשכוח שעם כל הכבוד לקומפוננטות, הן כתובות בטיפוסקריפט שמומר לג'אווסקריפט ומתנהגות כמו כל קוד אחר של טיפוסקריפט. אם אני אשים `console.log` בפונקציה של הקומפוננטה אני אוכל לראות את המשתנה בקונסולה של הדפדפן. אם אני אפתח את כל הפתחים, אני אכן אראה את הטקסט שעשיתי לו `export`:

```
src > Vars.js
1 export default 'Hello';

src > App.js
1 import React from 'react';
2 import logo from './logo.svg';
3 import './App.css';
4 import myVar from './Vars.js';
5
6 function App() {
7   console.log(myVar);
8   return (
9     <div className="App">
10       <header className="App-header">
11         <img src={logo} className="App-logo" alt="logo" />
12       </header>
13     </div>
14   );
15 }
```

אנחנו יכולים ליזא הכל. למשל, משתנה שיש בו טקסט או מספר:

```
const someVar:string = 'Hello';
```

```
export default someVar;
```

שימוש לב שבשלב זה אני לא חייב שהשם של המשתנה יהיה זהה ביצוא ובייבוא. בדוגמה זו אני מיצא משתנה שנקרא someVar, אבל מייבא אותו כ-myVar. אני יכול לקרוא לו בכל שם שאני רוצה כל עוד אני משתמש במילה השמורה default שעלייה נלמד בהמשך.

אנחנו יכולים לעשות export לפונקציה. למשל:

```
function foo():string {
    return 'Hello!';
}
export default foo;
```

כדי לשים לב שמדובר בפונקציה רגילה ולא בפונקציית קומפוננטה (רואים את זה כי הוא לא מתחילה באות גדולה ולא מחריזה JSX אלא מחרוזת טקסט). כשאני מיבא אותה (שוב, באיזה שם שאני רוצה) אני צריך להפעיל אותה בדרך כללshi. בדוגמה זו אני מבצע ייבוא בשורה השנייה  
ומستخدم בפונקציה זו:

```
import './App.css'
import myVar from './Vars.ts';

function App():JSX.Element {
  console.log(myVar());
  return (
    <>
      OUR CODE WILL BE HERE
    </>
  )
}

export default App
```

ואם זה נראה לכם מוכר, זו בדיקת הדריך לייבא וליצאת פונקציה שהיא קומפוננטה בריאקט. קומפוננטה בריאקט היא בסופו של דבר פונקציה ג'אוועסקרופיטית לכל דבר, רק צו שמשתמשת ב-XS. אנחנו מיבאים אותה ומשתמשים בה ב-XS (בניגוד לפונקציה רגילה, שלא אנו קוראים). זה הכל. כדי לשים לב סיום שם הקובץ במקרה של פונקציות שאין בהן קוד JSX יהיה ts ולא .tsx.

## ייבוא מנתיבים אחרים

עד עכשוו ייבאו קוד שנמצא בבדיקה באוטה תיוקיה של הקומפוננטה שבה השתמשנו בקוד המיווא. אבל מה קורה אם הקוד שלנו נמצא נמצאת בתיקייה אחרת? במקרה זה אנו צריכים להקליד את הנתיב היחסי של התיקייה שלנו ביחס לקובץ שבו אנו נמצאים.

אם למשל הקוד של myVars.ts, שלפי הסימות שלו אנו רואים שאין בו קומפוננטה ריאקטית, נמצא בתת-תיקייה בשם components, איני איבא אותו כך:

```
import myVar from './components/Vars.ts';
```

מהה כתובות זו מורכבות?  
 החלק הראשון - ./ מסמן את המיקום של התיקייה שלנו, הקובי שבו אני כותב את הקוד.  
 מהנקודה זו הכל מתייחס לנו צריכים לכתוב נתיב יחסית מהמקום שבו אנו נמצאים.  
 החלק השני – תת-התיקייה .components  
 החלק השלישי – שם הקובי כרגע. אם מדובר בסיום של ts היה לא הכרחי:

```

  MY-APP
    > node_modules
    > public
    < src
      < components
        JS Vars.js
        # App.css
        JS App.js
      M
    1 import React from 'react';
    2 import logo from './logo.svg';
    3 import './App.css';
    4 import myVar from './components/Vars.js';
    5
    6 function App() {
    7   console.log(myVar());
    8   return (
    9     <div className="App">
  
```

כמובן, אם יש כמה תיקיות אנחנו יכולים לבצע import עמוק ככל שנרצה. למשל:

```
import myVar from './foo/bar/baz/components/Vars.ts';
```

הענין מסתבכים כאשר התיקייה שלנו רוצים ליבא ממנה היא לא תת-תיקייה של התיקייה הנוכחית שאנחנו נמצא בה, אלא תיקייה שנמצאת מעל או תוקייה אחות. במקרה זהו אנו נעזרים ב-./. שאומרים "עלות תיקייה אחות לעלה".

למשל, הנה ננו שיש לאפליקציה שלנו מבנהתיקיות כזה:

```

src/
  └── components/
    └── MyVar.ts
  └── application/
    └── App.tsx
  
```

אני רוצה ש-App.tsx תייבא מ-MyVar.ts. איך אני עושים את זה? אני צריך "עלות" תיקייה אחות לעלה מ-App.tsx אל src ואז לרדת אל תיקייה components. את המסע הזה אני מתאר באמצעות import כך:

```
import myVar from './.../components/Vars.ts';
```

הבה נראה את המסע:  
 חלק א' – ./ אנו מתחילה מהמקום שלנו.  
 חלק ב' – ../ עולים תיקייה אחות ל-src.

חלק ג' – יורדים אל תיקיית `components`.  
 חלק ד' – קובץ `MyVar.ts`.

זה נראה מבלבל בהתחלה, אבל צריך לזכור שבמקרה של תיקיות מורכבות, זה מה שצורך לעשות. גם סביבת העבודה שלכם (Visual Studio Code) אמורה לסייע לכם באמצעות השלמה אוטומטית.

## אין חובה להשתמש בסימנת הקובץ `ts`

כברית מחדל, אין צורך להשתמש בסימנת `ts` של הקובץ. אפשר לעשות את היבוא באופן הבא:

```
import myVar from './components/Vars';
```

## יצוא של כמה משתנים

בריאקט יש לנו כלל של קומפוננטה אחת בקובץ אחד, אבל לעיתים יש צורך ביצוא של משתנים רבים באותו קובץ, למשל משתנים. אנחנו לא נשים משתנה אחד בכל קובץ. במקרה הזה ניפרד לשילום `default` וניצא יישורות את מה שאנו צריכים ליצא. למשל:

```
export const foo:string = 'Omri';
export const bar:string = 'Kfir';
export const baz:string = 'Daniel';
export const daz:string = 'Michal';
```

הפעם, כיוון שאין לנו `default`, אנו חייבים ליבא את מה שאנו רוצים בדוק בשם שבו אנו מיצאים אותו. איך? כך:

```
import { foo } from './Vars'
```

או:

```
import { foo, bar, baz } from './Vars'
```

אנחנו יכולים ליבא הכל כאובייקט ואז לקרוא למשתנים השונים כתוכנות של האובייקט באמצעות ייבוא של הכל – שימוש בסימן כוכבית (\*) ובמילה השמורה `as`. כשאני משתמש ב- `as *` אני נדרש לציין את תוכנות האובייקט שאני רוצה להביא. לשם הדוגמה במקרה שלנו:

```
import * as myImportedObject from './Vars'
```

והיבוא יlook כך:

```
import * as myImportedObject from './Vars'  
console.log(myImportedObject.foo);
```

## ייבוא של תמונות, CSS ומשאבים אחרים

אך על פי ש-import ו-export הם תקן של ג'אווהסקריפט, בפועל מי שדואג לטעינה של המשאבים היא Vite שעלייה אנו לומדים וכן במקרה מסוים הואב המודרניות הוא הספרייה וובפак (webpack). הספרייה הזו היא מודול מבוסס Node.js ואחרראית על תהליך הבילד (build) של האפליקציה, תהליך שגם במסגרת מתנהלת טעינת הקבצים.

אנו מסוגלים להשתמש ב-import גם כדי לייבא תמונות וקובצי CSS. אפשר לראות ב-`App.tsx`:  
המקור שמנגע כבירית מחדל עם Vite כיצד מיבאים תמונה וגם CSS:

```
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {

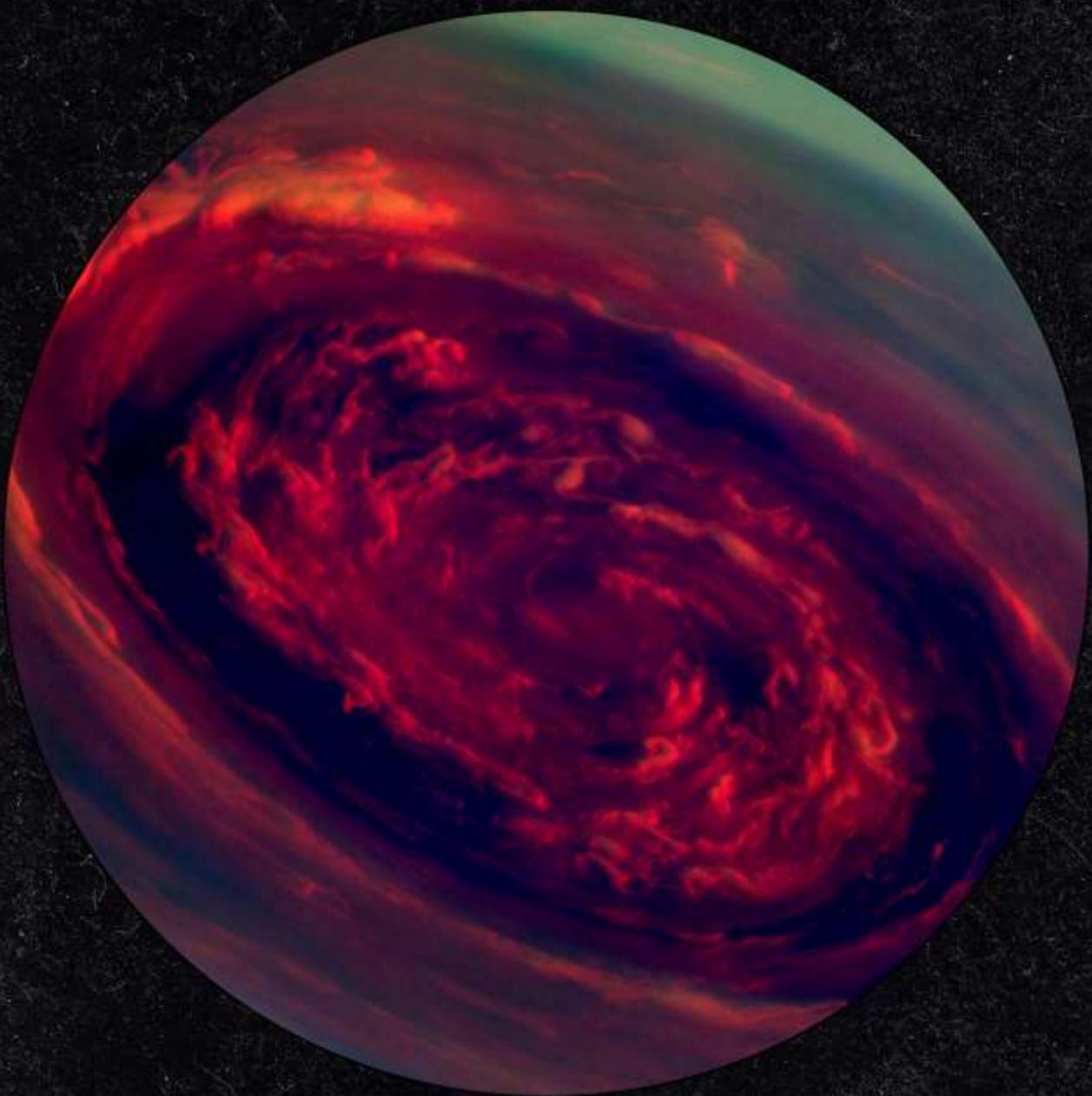
  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React
logo" />
        </a>
      </div>
    </>
  )
}

export default App
```

אני מיבא CSS וברגע שאני עושים את זה הוא מופעל ודואג לעיצוב ה-`JSX`/תמונה (ועל כך ארכיון בפרק הבא). את הייבוא והיצוא נתרגל בפרקים אחרים.

פרק 9

# JSX



# JSX

השימוש בטיפוסקריפט והסימת `ts` עלולה לבלב אותנו. המונח שמשמש אותנו בכל הנוגע לקומפוננטות הוא JSX. ג'אויסקריפט שרצ בריект. אנו משתמשים בטיפוסקריפט לשם פירוט סוג הميدע, אבל הקוד מומר ל-JS וاز לג'אויסקריפט רגיל.

עד כה לא התעמקנו כל כך ב-**JSX**, אותו HTML שנמצא בתוך `return` בקומפוננטה ובכל הדוגמאות. השתמשנו "סתם" ב-HTML. אך לא מדובר ב-HTML אלא ב-**JSX**, סוג של XML מונחה תגיות שמונע עם טיפוסקריפט. על מנת להדגים נשנה את `App.tsx` שמכיל את הקומפוננטה הראשית. כרגע יש שם HTML בלבד, אך אנו נתחילה לשחק איתו.

הכלל הוא פשוט – כל ביטוי שמופיע בין סוגרים מסוללים `{ }` ונמצא ב-**JSX** בעצם נחשב לג'אויסקריפט ורץ כג'אויסקריפט והתוצאה מופיעה בקומפוננטה. ככלומר, אם יש לי משתנים ואני רוצה להציג אותם, אני אכניס אותם ל-**JSX** עם סוגרים מסוללים:

```
import './App.css'

function App():JSX.Element {
  const myVar:string = 'Hello World!';
  return (
    <>
      {myVar}
    </>
  )
}

export default App
```

למשל, בדוגמה זו אני מכירז על משתנה `myVar` וublisher אותו ל-**TS**. הקומפוננטה כבר תדאג להציג אותו. אם אציג את הקומפוננטה באמצעות מבט בדף (בכתובת `localhost:5173`) אני אראה את הטקסט של המשתנה.  
ובן שאין יכול להציג סוגרים מסוללים כמו פעים שאין רוצה. למשל:

```
import logo from './logo.svg';
```

```
import './App.css';
import './App.css'

function App():JSX.Element {
  const myVar:string = 'Hello World!';
  const anotherVar:string = 'My Name is Ran';
  return (
    <>
      <p>{myVar}</p>
      <p>{anotherVar} {myVar}</p>
    </>
  )
}

export default App
```

שימוש בקוד זהה ידפיס את המשתנים במקום המתאים.

אבל JSX הוא הרבה יותר מהדפסה. כאמור, אנו יכולים להכניס JSX לתוך משתנה ולהדפיס אותו.  
למשל:

```
import './App.css'

function App():JSX.Element {
  const myVar:JSX.Element = <span>Hello World!</span>;
  return (
    <>
      {myVar}
    </>
  )
}

export default App
```

וכן, אני גם יכול לשים ב-JSX שנקנס לתוך משתנה משתנים בתוך סוגרים מסולסלים.  
העניין הוא ש-TSX הוא ג'אוوهסקרייפט טהור ויש להתייחס אליו בהתאם. אפשר לעשות אותו  
הכל בדיקות כמו בג'אוوهסקרייפט. למשל, להכניס בתוכו ביטוי:

```
<p>{ 1+1 }</p>
```

אם תציבו את ה-JSX זהה בקומפוננטה שלכם, תראו שהוא מופיע 2.

אבל הכוח של JSX הוא לא בתעלולים נלזים כאלה אלא במקומות אחרים. אפשר להשתמש בו בלוואות. למשל:

```
import './App.css'

function App():JSX.Element {
  const myChildren:string[] = ['Omri', 'Kfir', 'Daniel', 'Michal'];
  return (
    <>
      <p>{myChildren.map(child => <span> {child} </span>)}</p>
    </>
  )
}

export default App
```

מה אנחנו רואים כאן? ראשית הגדרת מערך של "הילדים שלי"

```
const myChildren:string[] = ['Omri', 'Kfir', 'Daniel', 'Michal'];
```

ניתן לראות שכנדרש בטיפוסקሪיפט הגדרתי את הסוג של המערך כמחרוזות טקסט. נכון, JSX נראה קצת זר ומזרר אבל הוא טיפוסקሪיפט והיכן שיש לי משתנים, קלט או פלט, אני חייב להגדיר את הסוג שלהם בטיפוסקሪיפט.

החלק השני הוא הגדרת סוגרים מסווגלים בתוך ה-`return` כדי שנוכל לעבוד עם ביטויי ג'אוوهסקרייפט.

```
<p>{ }</p>
```

בתוך הסוגרים האלה אנו יכולים לכתוב כל ביטוי ג'אוوهסקרייפטי שהוא. במקרה זה אנו נעשו `map` פשוט למערך שייחזיר לנו בכל פעם שם של ילד אחר.

```
myChildren.map(child => <span> {child} </span>)
```

בתוך ה-`map` יש לנו פונקציית חץ פשוטה שאנו מכירים ומחזירה גם היא JSX. על מנת שגם שם הילד יודפס בין ה-`span`, אנו מקיפים גם אותו בסוגרים מסווגלים, וזה כבר מתחילה להיות מעניין יותר.

אני יכול להשתמש בו במשפט תנאי. למשל:

```
import './App.css'

function App():JSX.Element {
  const someVar:boolean = true;
  return (
    <>
      { someVar === true ? <span>I am TRUE</span> : <span>I am
FALSE</span>}
    </>
  )
}

export default App
```

אם יש לי מערך של אלמנטים, אני יכול לזרוק אותם ישירות לתוך ה- JSX והוא כבר ידפיס לי אותם באופן אוטומטי.

```
import "./App.css";

function App(): JSX.Element {
  const myChildren: JSX.Element[] = [
    <li>Omri</li>,
    <li>Kfir</li>,
    <li>Daniel</li>,
    <li>Michal</li>,
  ];
  return (
    <>
      <ul>{myChildren}</ul>
    </>
  );
}

export default App;
```

כך למשל ייצורתי מערך של אלמנטים. כדי להדפיס אותו אני לא חייב להשתמש ב-map אלא רק לשימושו ב- JSX והוא יודפס אוטומטית. זה לא יעבוד עם אובייקט, רק עם מערך. חשוב לשים לב שמדובר הוא מערך של אלמנטים של JSX ולפיכך אנו מגדירים את הסוג שלו.

בעתיד, כשהנראה איך המשתנים האלו מגיעים מצד השרת או מקלט של המשתמש, הכל יעשה הרבה יותר מעניין, אבל כבר עכשיו זה אמרור לטלטל את עולמכם. עד כה אמרנו לנו לא לערраб HTML וג'אווהסקריפט. אבל חשוב לציין שהוא אולי נראה כמו HTML, אבל זה לא JSX. JSX שונה מ-HTML כי הוא ג'אווהסקריפט. כך, למשל, אם נכתבת המילה class, שהיא וליידית לחולtein ב-HTML אך מילה שמורה בג'אווהסקריפט, נקבל שגיאה.

לדוגמה, הקוד הזה:

```
function App(): JSX.Element {
  const myChildren: JSX.Element[] = [<li>Omri</li>, <li>Kfir</li>,
<li>Daniel</li>, <li>Michal</li>]
  return (
```

```

<div class="App">
  <header class="App-header">
    <img src={logo} class="App-logo" alt="logo" />
    <ul class="My-children" >
      {myChildren}
    </ul>
  </header>
</div>
);
}

```

לא יעבוד, כיוון שהמילה `class` היא שמורה. זו הסיבה שאנו משתמשים ב-`className`. גם קוד זה:

```

function App():JSX.Element {
  return (
    
  );
}

```

לא יעבוד, כי ב-`JSX` אנו חייבים להכניס אלמנט סגור. כך למשל:

```

function App():JSX.Element {
  return (
    
  );
}

```

הבדל נראה קטן אך הוא משמעותי. לא מדובר פה ב-`HTML` אלא ב-`JSX`, ג'אויסקרייפט שעבוד עם אובייקטי `XML`, שאלו גם האובייקטים ש-`HTML` עובד איתם.

## רשימות ב-`JSX`

אחד התסրיטים הנפוצים ב-`JSX` הוא המרת מערך לרשימה. למשל, מערך של שלושת האבות:

```
const fathers:string[] = ['Avraham', 'Itzhak', 'Yaakov'];
```

איך אני אdfsis אותו? באמצעות פונקציית map או לולאת for. מקובל מאד להשתמש בפונקציית map שעובדת על מערכים. במקרה של הדפסת רשימה כזו, ריאקט דורשת מאייתנו להוסיף לכל פריט ברשימה את התכונה key, שמכילה מזהה ייחודי, במקרה זה – המיקום של הפריט ברשימה, אם נרצה להציג את המערך כמו שצריך ברשימה זו:

```
const fathers:string[] = ['Avraham', 'Itzhak', 'Yaakov'];

const listItems:JSX.Element[] = fathers.map((father, index) =>
  <li key={index}>
    {father}
  </li>
);

return (
  <div className="App">
    <header className="App-header">
      <img src={logo} className="App-logo" alt="logo" />
      <ul>{listItems}</ul>
    </header>
  </div>
);
```

מה שחשוב פה הוא ה-key שמקבל את ה-index – קלומר המיקום של המשתנה באיבר. ההנחה המובלעת היא שהקוראים מכירים את פונקציית map ואת פונקציית `key`.

אם הקוד הזה:

```
const listItems:JSX.Element[] = fathers.map((father, index) =>
  <li key={index}>
    {father}
  </li>
);
```

נראה לכם לא מובן, כדאי למד שוב על פונקציית `map` ועל פונקציית `chz`, משום שהן קריטיות להבנה ומשתמשים בהן המון בריקט.

### תרגיל:

צרו קומפוננטה בשם `TodayTime` והציגו בה את התאריך העדכני.  
תזכורת: אנו מקבלים את התאריך העדכני בג'אווהסקריפט כז:

```
const today:string = new Date().toString();
```

### פתרון:

ראשית ניצור אתקובץ `TodayTime.tsx` בתיקיית `src` ובתוכו ניצור את הקומפוננטה שלנו:

```
function TodayTime():JSX.Element {
  const today:string = new Date().toString();
  return (
    <div>{today}</div>
  );
}

export default TodayTime;
```

הקומפוננטה הזו די פשוטה, אבל יש בה דבר אחד חשוב – אני מגדיר את התאריך של היום ואז מדפיס אותו באמצעות סוגרים מסולסלים.

את הקומפוננטה אני צריך (כלומר מיבא ומשתמש בה) כרגע איפה שאני רוצה. במקרה זה, ב-  
App:

```
import './App.css'
import TodayTime from './TodayTime';

function App() {
  return (
    <>
      <TodayTime />
    </>
  )
}

export default App
```

**תרגיל:**

הדףון, באמצעות לולאת for, את המספרים 10 עד 1 בתוך רשימת HTML. כלומר כך:

```
<ul>
  <li>10</li>
  <li>9</li>
  <li>8</li>
  <li>...</li>
  <li>1</li>

</ul>
```

**תזכורת:**

הדףון לולאת for מ-10 עד 0 עובדת כך:

```
for (let i = 10; i > 0; i--) {
```

```
}
```

**פתרונות:**

```
import "./App.css";

function App() {
  const items: JSX.Element[] = [];
  for (let i = 10; i > 0; i--) {
    items.push(<li key={i}>{i}</li>);
  }

  return (
    <>
      <ul>{items}</ul>
    </>
  );
}

export default App;
```

אנחנו יוצרים מערך של `items` ומאלסים אותו באמצעות לולאת `for` פשוטה. בכל איטרציה אנו משתמשים במתודת `push` שמכניסה איבר חדש למערך. מה היא מכניסה? JSX של אלמנט `<a>` עם ? נדיי לשים לב שכיוון שאנו רוצים שה-`i` חzieר את ערכו – אנו שמים אותו בסוגרים המסוללים. לא נשכח גם לשים `index` ב-`key`.

כל מה שנותר לנו לעשות הוא לחת את המערך ולהדפיס אותו. כאמור, JSX יודע לחת מערך ולטפל בו אוטומטית.

### תרגיל:

נתון אובייקט משתמש:

```
type UserInfo = {
  name: string,
  lastName: string,
  city: string,
  id: string,
}

const user:UserInfo = {
  name: 'Ran',
  lastName: 'Bar-Zik',
  city: 'Petah Tiqwa',
  id: '666',
}
```

הדפיסו את ה-`key` וה-`value` שלו כ-`JSX`, כך שהפלט הבא יופיע:

name: Ran lastName: Bar-Zik city: Petah Tiqwa id: 666

תזכורת: על מנת לקבל את כל המפתחות והערכים של אובייקט, צריך להריץ עליו את פונקציית האיטרציה `map` באופן הבא:

```
Object.entries(user).map(([key, value], index) => {  
    // Property order: index  
    // Property name: key  
    // Property value: value  
})
```

**פתרונות:**

```

import "./App.css";

function App() {
  type UserInfo = {
    name: string;
    lastName: string;
    city: string;
    id: string;
  };

  const user: UserInfo = {
    name: "Ran",
    lastName: "Bar-Zik",
    city: "Petah Tiqwa",
    id: "666",
  };
  const userArray: JSX.Element[] = [];
  Object.entries(user).map(([key, value], index) => {
    userArray.push(
      <span key={index}>
        {key}: {value}
      </span>
    );
  });
  return <>{userArray}</>;
}

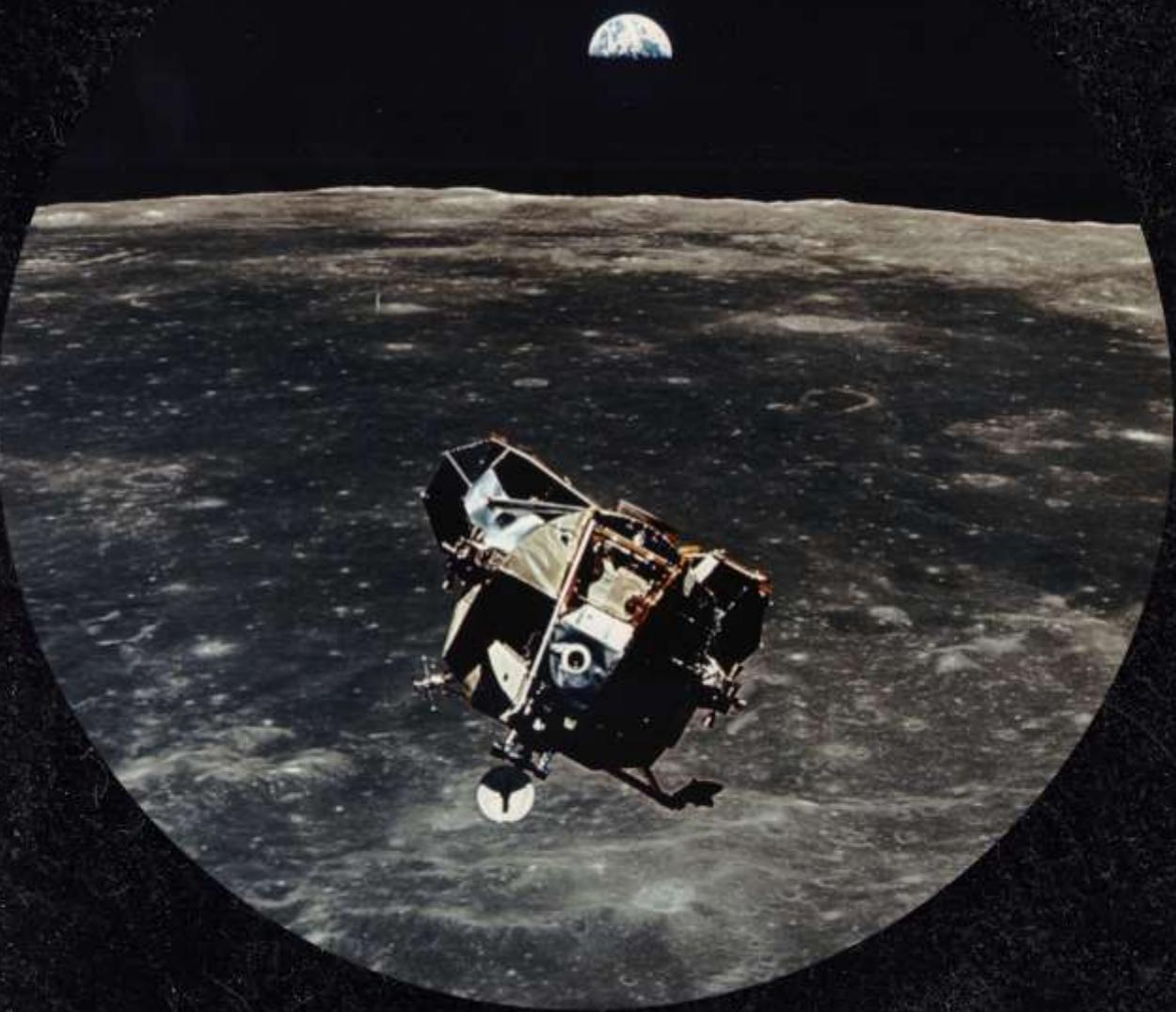
export default App;

```

ראשית, הדקתי את האובייקט שנדרש ממנו כדי לפתח בתרגיל. שנית, יצרתי פונקציית `map` ואת ה-`JSX` והכנסתי אותו לתוך מערך באמצעות `push`. השלב הבא הוא פשוט לשים את המערך זהה בתוך ה-`JSX` ולסמן עליו שירננדר הכל.

פרק 7

# הומפוננטה עם תוכנות (PROPS)



# קומפוננטה עם תכונות (props)

הקומפוננטות שבנו עד כה היו קומפוננטות סטטיות וטיפשות למדי, הן לא עשו הרבה חוץ מלהדפיס פלט. אבל במקום שיש פלט יש גם קלט – היכולת שלנו לשלוח נתון לקומפוננטה ובהתאם לו לקבל פלט. כך, למשל, אני יכול לשולח שם והקומפוננטה תחזיר לי: "בוקר טוב, רן" או "ערב טוב, רן" בהתאם לשם שאני שולח לה. אני יכול ליצור קומפוננטת טבלה ולשלוח לה מידע בפורמט JSON כדי שהקומפוננטה תציג את המידע בפורמט טבלאי. אני יכול ליצור קומפוננטת טופס שיש בו פרטיהם וכו'. הקלט הוא קריטי בפייתו מונחה קומפוננטה ובדרך כלל נמצאת קומפוננטות ללא קלט רק בתחילת שרשרת הקומפוננטות שלנו.

העברת קלט לקומפוננטה נעשית בריект באמצעות `props`, תכונות שאנו מעבירים אל הקומפוננטה שלנו ומהוות את הקלט. הנה נראה זאת באמצעות קומפוננטה שנקראת `Welcome`. הקלט שאנו רוצים להעביר הוא השם של המשתמש והפלט הוא "שלום, [שם]".

ראשית, ניצור את הקומפוננטה הבסיסית בקובץ `Welcome.tsx`.

```
function Welcome(): JSX.Element {
  return (
    <span></span> // Component will be here
  );
}

export default Welcome;
```

עכשו נוסיף את הטקסט שלנו:

```
function Welcome():JSX.Element {
  const name:string = 'Ran';
  return (
    <span>Hello, {name}!</span>// Component will be here
  );
}

export default Welcome;
```

ואם אני-arצה לדרוש את הקומפוננטה הזו, אני-arצה אותה עם import (שים לב שאלה-arצה לשנות את הנתיב, כפי שלמדו בפרק על import ו-export):

```
import Welcome from './Welcome';
```

ואשתמש בה בכל מקום פה:

```
<Welcome />
```

מה הבעה? היא שאני רוצה להעביר את השם כפרמטר. יכול להיות שבדף שבו אני משתמש בקומפוננטה הזו יש לי כל המידע שאני צריך – אז איך אני מעביר את השם לקומפוננטה? אני יוצר props. ראשית, בקומפוננטה אני אכנים props כפרמטר לפונקציה. זה ייראה כך:

```
function Welcome(props)
```

ה-props זהה הוא בעצם אובייקט הקלט שלנו. יש בו כל הפרמטרים שנעביר לקומפוננטה. אין לנו מעבירים? באמצעות התוכנה של הקומפוננטה, שנראית בדיקת כמו תוכנה של HTML. הנה, כן:

```
<Welcome name="Moshe" />
```

כאן העברנו קלט בשם name. איך נקבל אותו? כך:

```
type propsInfo = {
    name: string,
};

function Welcome(props: propsInfo): JSX.Element {
    const name = props.name;
    return (
        <span>Hello, {name}!</span> // Component will be here
    );
}

export default Welcome;
```

כלומר אנו מعتبرים פרמטרים באמצעות תכונות. תכונות של תגיוט HTML נקראות באנגלית attributes HTML. למי שלא מכיר כל כך HTML, תכונות הן בעצם הדרך שלנו להעביר מידע לתגית HTML כמו קישור. למשל:

```
<a title="Books" href="https://hebdevbook.com">Books</a>
```

ה-href וה-title נקראים "תכונות", וזה הדרך שלנו לתקשר עם רכיבי HTML ולשנות אותם. אותו הדבר עם קומפוננטות ריאקטיות. באמצעות שימוש בתכונות אנו מعتبرים להן מידע. אנו נגדיר גם את סוג המידע של התכונות עם טיפ נמקובל בטיפוסקריפט. אם שכתבם אין יוצרים type ומה הוא אומר – זה הזמן לבצע חזרה באמצעות החלק על טיפוסקריפט בספר "לימוד ג'אויסוסקריפט בעברית". בו יש הסבר מפורט על סוג מידע. אנו חייבים להגיד סוג מידע שהוא מוגדר בקובץ config.json. אם לא, Vite יסרב לעבוד. הגדרת סוג מידע חשובה כיון שהיא מונעת מצפים לקבל בקומפוננטה. אם לא, Vite יסרב לעבוד. הגדרת סוג מידע חשובה כיון שהיא מונעת תקלות ובעיות בהעברת מידע לקומפוננטות שאחריהם כתובים.

הבה נדגים שוב. נניח שבאותה קומפוננטה אני רוצה להעביר את התואר של האדם, למשל Mr. או Doctor, וכך אוכל לברך אותו. איך אני יכול לעשות זהה דבר? בקומפוננטה שלי אני אגדיר את הדרך שבה שולחים לי את התואר, למשל prefix ו商量ון את סוג המידע, שהוא מחרוזת טקסט:

```
type propsInfo = {
  name: string,
  prefix: string
};

function Welcome(props: propsInfo): JSX.Element{
  const name:string = props.name;
  const prefix:string = props.prefix;
  return (
    <span>Hello, {prefix} {name}!</span> // Component will be here
  );
}

export default Welcome;
```

וכמובן, זה נשלח עם props. איך אני שולח? כשאני משתמש בקומפוננטה:

```
<Welcome name="Moshe" prefix="Doctor" />
```

ומה אראה?

Hello, Doctor Moshe

הכוח האמתי של ה-props הוא לא במשЛОוח מחרוזות טקסט. אני יכול לשלווח הכל. ב-JSX אני יכול לשלווח למשל אובייקטים שלמים. בואו נניח שיש לי אובייקט משתמש עם שם ותואר, שהוא זהה:

```
const user = {
  name: 'Moshe',
  prefix: 'Doctor',
}
```

איך אני שולח את האובייקט זהה?

אני יכול לעשות משהו כזה:

```
<Welcome name={user.name} prefix={user.prefix} />
```

אני משתמש ב-{} כדי להעביר את הפרמטרים ב-JSX.

אני יכול לעשות משהו אחר – למשל לשנות את הקומפוננטה שלי, כך שתדע לקבל אובייקט.  
למשל:

```
type propsInfo = {
  user: {
    name: string,
    prefix: string,
  };
}

function Welcome(props:propsInfo):JSX.Element {
  const name:string = props.user.name;
  const prefix:string = props.user.prefix;
  return (
    <span>Hello, {prefix} {name}!</span> // Component will be here
  );
}

export default Welcome;
```

דרך נוספת היא להשתמש ב-destructuring. מדובר בכך מיוחדת שבה אנו לוקחים אובייקט ומפרקים אותו למערך. אני מעביר את האובייקט שלי ב-prop אחד:

```
type userInfo = {
  name: string,
  prefix: string,
};

function App(): JSX.Element {

  const user: userInfo = {
    name: "Moshe",
    prefix: "Doctor",
  }
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <Welcome user={user} />
      </header>
    </div>
  );
}
```

כשאני רוצה להשתמש במסתנים השונים אני מבצע הlion מיוחד שנקרא destructuring. אני בזמנים לוקח את האובייקט וממיר אותו למשתנים. שם המשתנה חייב להיות זהה לשם האובייקט. למשל:

```
type propsInfo = {
  user: {
    name: string,
    prefix: string,
  };
}

function Welcome(props:propsInfo):JSX.Element {
  const { prefix, name } = props.user;
  return (
    <span>Hello, {prefix} {name}!</span> // Component will be here
  );
}

export default Welcome;
```

מה קורה פה? פשוט מאד. האובייקט של `props.user` מורכב נזכר גם `prefix` וגם `name`. במקום לעשות מהו זה:

```
const name = props.name;
const prefix = props.prefix;
```

אני עושה מהו זה:

```
const { prefix, name } = props.user;
```

זה בדוק אותו הדבר.

כרגע אזכיר, זה הדבר הכי חשוב בקומפוננטה, לקבל פרמטרים פנימה. זה הקלט שלנו ואיתו אנו

עובדים. הפלט? הפלט במקרה הזה הוא מה שנוצר כתוצאה מrendor ומה שהוא רואים. בהמשך

נלמד על כך שהפלט האמיתי הוא האירועים שמופעלים בקומפוננטה.

**תרגיל:**

צרו קומפוננטה בשם Birthday המקבלת מספר באמצעות תכונת age ומציינה אותו בטקסט באופן הבא:

Happy Birthday! You are X years old!

כש-X הוא הגיל. קראו לקומפוננטה זו m-app.

**פתרון:**

הקומפוננטה נשמרה בשם `age.tsx` והוא מכילה את הקוד הזה:

```
type propsInfo = {
    age: number,
}

function Birthday(props:propsInfo):JSX.Element {
    const age = props.age;
    return (
        <span>Happy Birthday! You are {age} years old!!</span>
    );
}

export default Birthday;
```

ב-`app` היא תיקרא כך:

```
<Birthday age={10} />
```

אפשר לראות שהדבר היחיד שמעניין כאן הוא שיש לי `props` ומהם אני לוקח את ה-`age`. כשאני קורא לקומפוננטה זו, אני מעביר לה `age`, בדוק כפי שלמדנו. אני גם מגדיר את סוג המידע של הגיל כמספר, נמקובל בטיפוס קריפט.

**תרגום:**

עדכנו את הקומפוננטה הקודמת כדי שתתקבל גם שם כתכונת name, והשם יופיע כך:

Happy Birthday Y! You are X years old!

X יהיה גיל ו-Y יהיה השם. הקריאה לקומפוננטה מ-App אמורה להיראות כך:

```
<Birthday name='Moshe' age={10} />
```

**פתרון:**

:Birthday קומפוננטה

```
type propsInfo = {
  age: number,
  name: string,
}

function Birthday(props:propsInfo):JSX.Element {
  const { age, name } = props;
  return (
    <span>Happy Birthday {name}! You are {age} years old!!</span>
  );
}

export default Birthday;
```

מה מעניין כאן? השתמשתי ב- destructuring כדי לקבל את התכונות שהועברו לקומפוננטה שלי.

הסינטקס:

```
const { age, name } = props;
```

הוא בדיק כמו:

```
const age = props.age;
const name = props.name;
```

אבל השימוש ב- **destructuring** אלגנטי יותר.

### תרגיל:

הארכיטקט הבכיר בחברה קובע שהkomponennta שלכם קיבל את האובייקט של ה-user, בסגנון זהה:

```
type userInfo = {
  name: string,
  age: number,
}

function App():JSX.Element {
  const user:userInfo = {
    name: 'Moshe',
    age: 10,
  }
  return (
    <div className="App">
      <header className="App-header">
        <Birthday user={user} />
      </header>
    </div>
  );
}
```

עדכנו את komponennta Birthday על מנת לעבוד עם ה-API החדש.

**פתרונות:**

כיוון שכפו علينا שינוי, אנו חייבים לעבוד עם האובייקט. אם באמת השתמשתם בו-, **destructuring**, יהיה לכם הרבה יותר קל לעבוד. צריך רק להבין שבסמךם `props.name` ו-`props.user` יש לנו `props.age`:

```
type propsInfo = {  
    age: number,  
    name: string,  
}  
  
function Birthday(props:propsInfo):JSX.Element {  
    const { age, name } = props.user;  
    return (  
        <span>Happy Birthday {name}! You are {age} years  
old!!</span>  
    );  
}  
  
export default Birthday;
```

**תרגיל:**

מנהל הפרויקט ביקש מכם לדאוג שהຄומפוננטה תציג בנוסף גם את הטקסט: You are 18 ומטה, או! You are OK!, אם גיל המשתמש הוא מעל 18.underaged!

**פתרונות:**

אסור לנו לשוכח שאף על פי שב-XSL עסקינו, עדין מדובר בטיפוסקריפט פשוטה וקלת, שאנו מכירים. אם אתם מתכנתים, אתם אמורים לשלוט במשפטי תנאי או במשפטי תנאי מקוצרם. הדרכ להכניס קבוע phrase טקסט לפי הגיל הוא באמצעות תנאי מקוצר:

```
const phrase:string = age <= 18 ? 'You are underaged!' : 'You are OK!';
```

או באמצעות משפט תנאי פשוט יותר אך אלגנטי יותר:

```
let phrase:string;
if (age <= 18) {
  phrase = 'You are underaged!';
} else {
  phrase = 'You are OK!';
}
```

לא משנה באיזו דרך בחרתם, אתם יכולים להשתמש במשתנה אחת כדי ליצור משתנה אחר ולשים אותו במה שהקומפוננטה מחזירה:

```
type propsInfo = {
  user: {
    age: number;
    name: string;
  };
}

function Birthday(props:propsInfo):JSX.Element {
  const { age, name } = props.user;
  const phrase:string = age <= 18 ? 'You are underaged!' : 'You are
OK!';
  return (
    <span>Happy Birthday {name}! You are {age} years old!
{phrase}</span>
  );
}

export default Birthday;
```

זה הכל. שוב, עם כל הכבוד ל- **JSX**  ולסינטקס המבלבל – זה בסופו של דבר ג'אוועסקרייפט.

פרק 8

# דיבאת



## דיבאג

חלק שימושתי מכל פיתוח תוכנה הוא הילך הדיבאגינג – קלומר מציאות התקלות, השגיאות והבעיות שיש בקוד שאנו מרכיבים. דיבאגינג פירושו מציאות באגים, וכשאנו כותבים קוד תמיד, אבל תמיד, יהיו בעיות.

בדומה לג'אווהסקריפט, אנחנו יכולים לבצע דיבאג לכל קומפוננטה ריאקט עם פירופוקס או כרום בקלות ובייעילות כשהאנו מרכיבים את סביבת הפיתוח שלנו. בנוסף על כן, יש לכרום ולפירופוקס "React" של כל מפתחים ייעודי לריאקט שאותו כדאי להכיר ולהתקין עכשו. חפשו "React Developer Tools"

לכרום <http://bit.ly/reactdevtool>

או על:

<https://addons.mozilla.org/en-US/firefox/addon/react-devtools/> לפירופוקס.

Home > Extensions > React Developer Tools



### React Developer Tools

Offered by: Facebook

★ ★ ★ ★ ★ 1,195

| [Developer Tools](#)

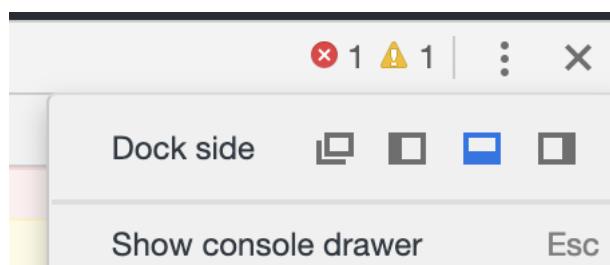
|  1,833,536 users

על מנת להדגים את תהליך הדיבאג, נחוץ לנו קוד לדבג. אתם יכולים להשתמש בכל קוד שכתבתם עד כה. בפרק זהה אני אדגים על הקומפוננטה שיצרנו בתרגיל בפרק הקודם. אבל התהליך עובד, כמובן, בכל קומפוננטה שהוא ובכל קוד שהוא. הדיבאגר המובנה בדף והוסף המינוח שמתלווה אליו אמורים להיות ידידיכם הטוביים ביותר.

ראשית, הדיבאגר של הדפדפן עובד כרגיל. בכל דףדף מודרני יש כלי מפתחים מובנה ובפרק זה נלמד על כרום, הדפדפן הנפוץ היום. אך יש לנו מפתחים זהה בכל דףדף עם אותו מבנה. בכרום אנו פותחים את כל הamentiils באמצעות צירוף המKeySpecים קונטROL, שיפט ו-**ו**-בחולנות או בלינוקס וקומנד, אופשן ו-**ו** במק. יפתח לכם כל הamentiils בתחום המסנן.

היכנסו אל האפליקציה של Vite והריצו אותה באמצעות dev run npm. לאחר שהאתר נפתח, פתחו את כל הamentiils.

**טיפ:** אפשר לשנות את מקום החלון באמצעות הקלדה על שלוש הנקודות ושינויו ה-**Dock side**:

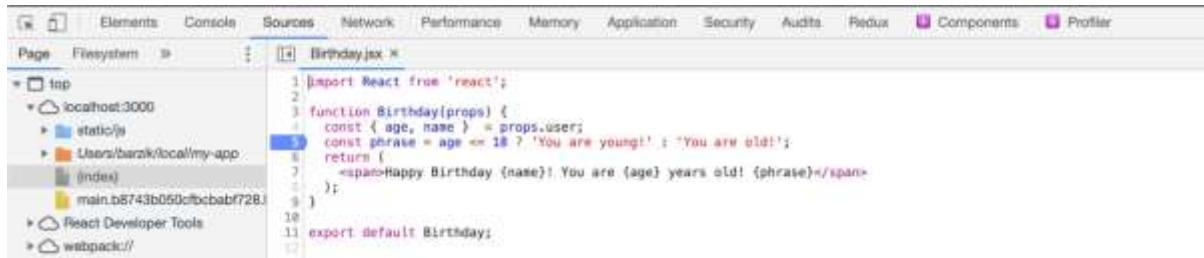


נעבור לתגית ה-**Sources**:



אנו יכולים לראות את מבנה האפליקציה שלנו והקומפוננטות לצד שמאל. אם אנו רוצים לחפש קובץ מסוים, לחיצה על קונטROL ו-**ו** בחולנות ובלינוקס או קומנד ו-**ו** במק תפתח תפריט חיפוש.

אנו יכולים להקליד את שם הקובץ של הקומפוננטה, למשל `xs/Birthday.tsx`, שיצרנו בתרגיל בפרק הקודם. בואו נראה את מבנה הקומפוננטה:

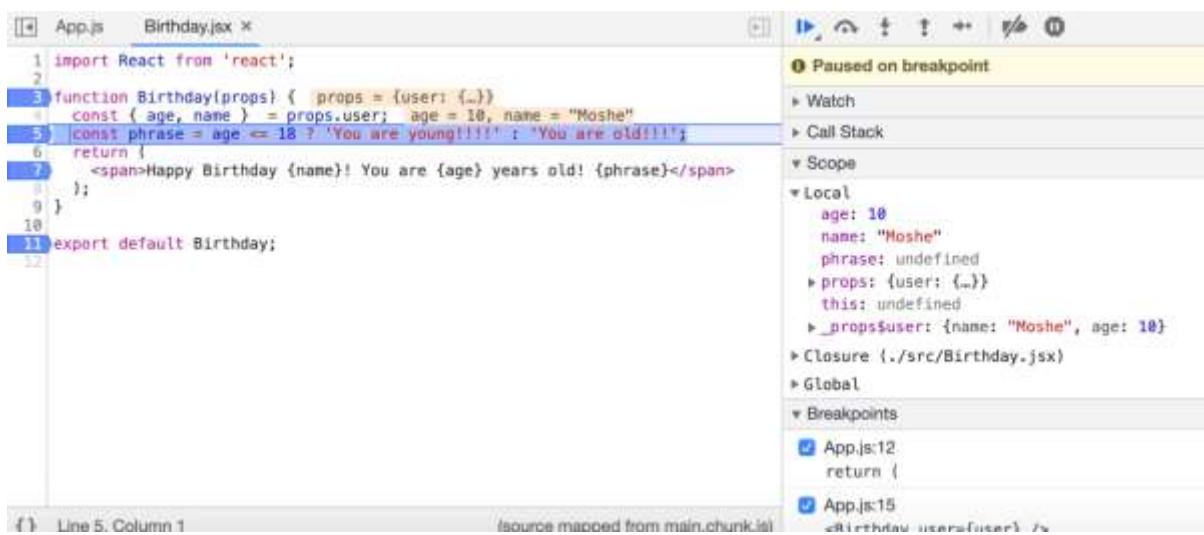


```

1 import React from 'react';
2
3 function Birthday(props) {
4   const { age, name } = props.user;
5   const phrase = age <= 18 ? 'You are young!' : 'You are old!';
6   return (
7     <span>Happy Birthday {name}! You are {age} years old! {phrase}</span>
8   );
9 }
10
11 export default Birthday;
12

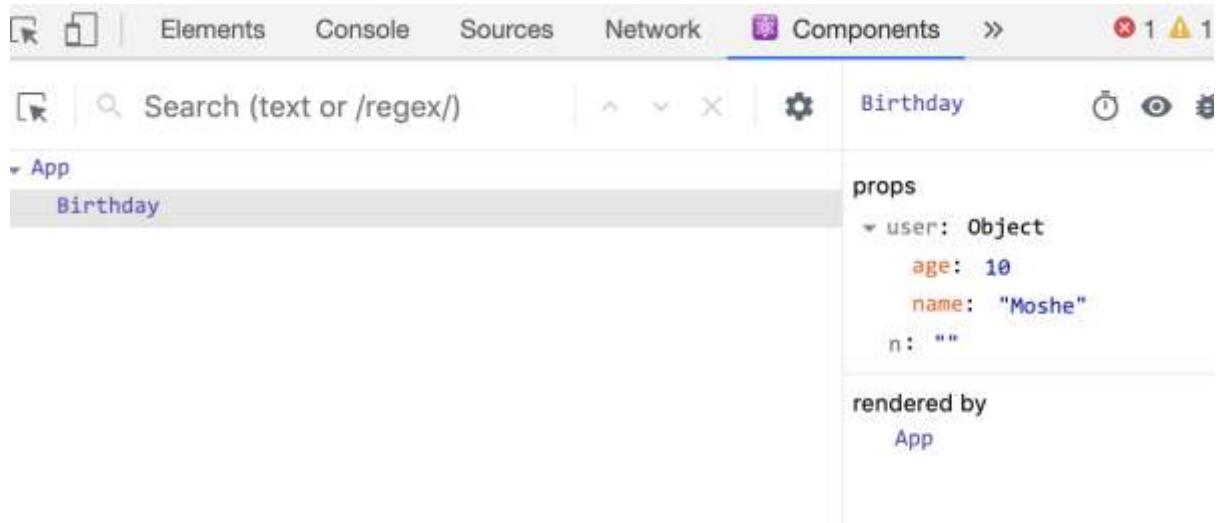
```

אנו יכולים, באמצעות לחיצה בצד השורה המתאימה, ליצור נקודת עצירה (Breakpoint). מדובר בנקודת שבת הסקריפט שלנו עוזר ולא ממשיך הלאה ואנו יכולים לראות מה מצב הקוד, לבדוק וראות את ערכי המשתנים ולאחר מכן המשיך את הרצאה. אם נעה מחדש את העמוד (באמצעות F5 למשל או כונטרול + z) התוכנה תבצע נקודת העצירה הזו ונוכל לראות את ערכי המשתנים, להקליד בקונסולה ולצעוד קדימה בזיהירות כדי להבין מה השتبש:



אפשר גם ליצור נקודות עצירה באפליקציית הריאקט שנמצאת ב-`app`, בכל קובץ שהוא. חשוב לציין שריект בסופו של דבר היא ג'אנושקריפט. לא יותר, לא פחות.

אך התקנת נוספת הפיתוח של ריאקט פותחת לנו אפשרות נוספת. החשובה ביותר היא צפייה ברשימה כל הקומפוננטות. אם התקנתם את תוסף הפיתוח, תוכלו לראות לשונית נוספת בשם `Components`. לחיצה עליה תציג לכם את רשימת כל הקומפוננטות. בתוך כל קומפוננטה תוכלו לראות את מצב `theProps` שלה ובאייזו אפליקציית ריאקט היא נמצאת:



לחיצה על איקון העין תמקד אתכם באלמנט הקומפוננטה כפי שהוא נמצא ב-HTML, ולהזיכה על האיקון של הבאג תציג את המידע של הקומפוננטה בקונסולה.

להרבה מפתחים מתחילה יש נטייה להימנע מהדיבagog כיון שמדובר בכלים שנראוה מפחיד וגם אין הרבה מה ללמידה – אבל זה מצער. כדאי לשאקו כמה שעות בניסיון להבין איך הוא עובד ולהשתמש בו במקומות לנסوت לפענה מה הבעיה בעצמכם או בעזרתו כל מיני מעקפים כמו console.log. זה הרבה יותר קל ומהיר, ובטוח האורך – משלtam מבחינת הזמן. כל המפתחים משתמשים ומשתפרים כל הזמן, וכך גם התוסף של ריאקט. יש סיכוי שזמן שתקרוו את הפרק, התצוגה של התוסף או של כל המפתחים תשתנה – הם עוברים שינויים כל הזמן. אבל הפונקציונליות הבסיסית של כל המפתחים לא משתנה – מקום לשינויים נקודת עצירה, להרייך קדימה ולראות מה מצב ה-props. אלו הדברים החשובים לכם.

מבחינת תרגול – נותר עליו בפרק הזה. התרגול שלכם יהיה בשימוש בדיAGER ב פרקים הבאים.

פרק 9

# סְנִירִים



# סטוּטִיט

עד כה יצא לנו לעבוד עם קומפוננטות סטטיות בלבד, ככלומר במקרה שיש בהן רק `return` של JSX או שמקבלות `props` ואז מוחזרות JSX. קומפוננטות אלו הן מצוינות אבל "חסרות זיכרון". ככלומר, אני מעביר לקומפוננטה מידע, היא מרנדרת אותו וזהו. גם אם אני אשנה את המידע, אחרי שהקומפוננטה הוצגה, אני עדיין אראה את המידע שהכנסתי בהתחלה.

אדגים את הבעיה עם קומפוננטה שנקראת `CountUp.tsx`. זהה קומפוננטה שעשושה משהו פשוט מאוד - היא מציגה כפטור ובכל פעם שאני לחץ על הcpfator, הספרה עולה ב-1. איך אני ממש דבר זהה בג'אוויסקייפט? בקלי קלות עם אירוע שצמוד לכפטור ויפעל פונקציה כזו:

```
let count:number = 0;
function countUp():void {
    count = count + 1;
}
```

לא משווה מרכיב מדי. שימוש לב שכיוון שהפונקציה לא מחזירה דבר, סוג המידע שאינו מפרט עבורה הוא `void`.

אני יכול להכניס את הקוד הזה בקומפוננטה באופן הבא:

```

function CountButton():JSX.Element {
  let count:number = 0;
  function countUp():void {
    count = count + 1;
  }
  return (
    <div>
      <h2>{count}</h2>
      <button onClick={countUp}>Click me</button>
    </div>
  );
}
export default CountButton;

```

**שימוש לב:** אין בעיה להשתמש בפונקציה בתוך פונקציה. הפונקציה שהגדנו בתוך הפונקציה תהיה זמינה אך ורק בתחום הסcop של הפונקציה.

**שימוש לב:** הפונקציה שאנו מפעילים ב-`onClick` מוקפת גם היא בסוגרים מסולסים כיון שאנו רוצים ששם יתורגם לקוד. אנו נמצאים ב-`JSX` וצריכים להבהיר שמדובר בפונקציה של הקומפוננטה.

אם אני אקרא לקומפוננטה זו ב-`app` באופן הבא:

```

import './App.css'
import CountButton from './CountButton';

function App() {
  return (
    <>
      <CountButton />
    </>
  )
}

export default App

```

אני אראה שכאן הקומפוננטה מתרננרט ומוצגת הספרה 0, אך הספרה לא מתקדמת בלחיצת ה-`count`. אם תציגו בדיAGER, שעליו למדנו בפרק הקודם, תוכלו לראות שכאן הפונקציה `Up` נקראת בכל לחיצה, אבל המספר לא מתעדכן. מדוע? כדי להבין את זה לעומק אנו צריכים להבין יותר מהו רנדור.

## ריאקט ורנדור

כאשר אנו כתבים קומפוננטה ב-`JSX` של האפליקציה שלנו, אנחנו לא מייצרים `HTML`, אלא אובייקט ריאקטי שריאקט מכניסה ומציגו אותו ב-`DOM` האמיתי. פועלות הציגה זו, שמתרגמת את `JSX` לתצוגת `HTML` שהדף יודע לטפל בה, נקראת "רנדור" `render` – הרצה. הרנדור הראשוני של כל קומפוננטה מתבצע בטעינה הראשונית ואנו רואים את התוצאה שלו בדף. כל עוד אין רנדור חדש, התצוגה בקומפוננטה לא תשתנה. זה בדוק מה שקרה עם הקומפוננטה `CountButton`. הפונקציה שנקרה מהכפתור אכן עובדת ומעדכנת את המשתנה הפנימי, אבל כל עוד אין רנדור חדש, נראה את זה בדף גם אם המשתנה הפנימי השתנה. על מנת להציג את השינויים שנעשו בקומפוננטה אנו חייבים לבצע רנדור חדש. הרנדור החדש נעשה באמצעות אחת משתי דרכים:

1. **שינויי חיצוני של `props`**, כלומר קומפוננטה אב משנה את `props` של הקומפוננטה. שינוי זה גורם לרנדור חדש מייד. בהמשך נלמד איך עושים את זה עם אירופים.
2. **שינויי הסטייט של הקומפוננטה**.

ברגע שיש רנדור חדש, הקומפוננטה מתעדכנת וגם `JSX`, ואני רואים את הערכים החדשניים ב-`JSX`. בדוגמה שלנו, אם אנו רוצים שהקומפוננטה תציג את הערך של `count` שהשתנה, אנחנו צריכים לדאוג לשנות את הסטייט הפנימי שלה. בשביל זה אנחנו צריכים לדעת מה הוא סטייט.

## סטייט

סטייט הוא בעצם הזיכרון הפנימי של הקומפוננטה. לקומפוננטות שעבדנו עד כה לא היה זיכרון. או שהן קיבלו את המידע שלהם מבפנים (קומפוננטות תצוגה) או שהן קיבלו את המידע שלהם מקומפוננטה אב דרך `props`. כך או אחרת, הן קיבלו את הקטל, עשו איתו חישוב והעיבו

אותו ל-**JSX**. הקומפוננטה רונדרה, ה-**JSX** הוצג בתור **HTML** ובעצם הסיפור הסיפור הסתיים. פה אנו זוקקים לזכור פנימי שיאפשר לקומפוננטה לנצל את המשתנים שלה, וחשוב מכך – לסמן לריאקט متى בדוק לרנדר את הקומפוננטה מחדש. הסטייט במרקחה של ריאקט הוא אובייקט שאנו מכנים אליו את המידע שאנו רוצים "לזכור". כאשר אנו מנסים את המידע שיש בזיכרון, יש לנו רנדור חדש.

از איך מנהלים את הסטייט? בקומפוננטות מסווג פונקציות אלו משתמשים במנגןון פשוט שנקרו **הוקים (hooks)** או ביחיד **hook**. בהוק אנו מבצעים שתי פעולות:

1. הגדרת החלק שאנו מכנים לסטיט.
2. הגדרת הפונקציה שבאמצעותה אנו מנסים את החלק זהה בסטייט. בקריאה לפונקציה זו מתבצע הרנדור מחדש.

זה נעשה באמצעות **hook useState**. הוק הוא שם מפheid לפונקציה פשוטה שמקבלת פרמטר אחד ויחיד – הערך ההתחלתי של הסטייט שלו. הפונקציה מחזירה מערך עם שני חלקים. החלק הראשון הוא הסטייט עצמו, שבו אני יכול להשתמש ב-**JSX** (או בכל מקום בקומפוננטה), והוא לקרוא בלבד. החלק השני הוא הפונקציה שבאמצעותה אני משנה את הסטייט. בטיפוסקריפט ניתן גם נדרש להגדיר את סוג המידע של הסטייט. **useState** בטיפוסקריפט מאפשר לי באמצעות גנריות להעביר את סוג המידע הזה. להזכירם, בטיפוסקריפט יש פונקציות התומכות בהעברת סוג המידע שהן אמורות להחזיר וזה נעשה עם חיצים מושלמים והגדרת סוג המידע.

ראשית נראה ואז נסביר. המטרה שלנו היא להכניס את המשתנה **count** לזיכרון של הקומפוננטה. אחרי שעשינו את זה, כל מה שנותר לנו לעשות הוא לשנות את **count** אך ורק בעזרת פונקציה מיוחדת לשינוי. הקומפוננטה המלאה נראה כך:

```
import { useState } from 'react';
function CountButton():JSX.Element {
  const [count, setCount] = useState<number>(0);
  function countUp():void {
    setCount(count + 1);
  }
}
```

```

return (
  <div>
    <h2>{count}</h2>
    <button onClick={countUp}>Click me</button>
  </div>
);
}

export default CountButton;

```

ראשית, אני קורא להוקים באמצעות import. זהה משתמש ב- destructuring של מדרנו עליו בפרקם קודמים. קיבלו אותו כמוות שהוא - בכלים אלה צריך להשתמש כאשר אנו נעזרים בהוקים.

הצעד השני הוא, כפי שהסבירנו קודם, להשתמש בפונקציית useState. אני מעביר לה את הערך ההתחלתי (0) ומתקבל מערך. האיבר הראשון הוא קבוע המיצג את ערך הסטיט, האיבר השני הוא הפונקציה שמשנה אותו. מקובל (אך לא חובה) לכתוב את שם הפונקציה כך שיתחיל ב-set ואז יבוא שם הסטיט באות גדולות.

למשל, אם המשתנה שרציתי להכניס לזכרון הוא count, שם הפונקציה שמשנה את המשתנה בזיכרון של הקומפוננטה יהיה `setCount`. במקרה שלנו שם המשתנה הוא: count, אך שם הפונקציה שמשנה את המשתנה בזיכרון יהיה `setCount`. זה הכלול.

אני גם צריך להגיד את סוג המידע של המשתנה שיש ב-state. אני עושים את זה עם גנריות שהסינטקס שלו הוא סוגרים משולשים וסוג המידע שאני מצפה לקבל מuseState. במקרה הזה מספר.

אני מבצע את ההגדלה זו באמצעות השורה:

```
const [count, setCount] = useState<number>(0);
```

הפונקציה המובנית useState היא הפונקציה הקרייטית פה, כיוון שהיא מקבלת את הערך הראשוני של המשתנה שלי, במקרה זה 0.

עכשו אני צריך לשנות בקומפוננטה ולהמיר את כל הפעמים שבהן אני משנה את הערך של `z` בפונקציה של `setCount`, הפונקציה שאומרת בעצם – תכניס לזכור את הערך הזה, במקרה שלנו – `:count + 1`:

`setCount(count + 1);`

זה הכל!

הבה נסקור את הפעולות שעשינו כדי ליצור "זיכרון" לקומפוננטה:

שם הפעולה	הקוד הישן	הקוד החדש
להביא את ההוקים		Import { useState } from 'react';
לקבוע ערך ראשוני לסתיט ולקבל את המשתנה לזכור, לקבל פונקציה שמשנה אותו		const [count, setCount] = useState(0);
להגידו ל-state את סוג המידע שאנחנו מוכנים שהוא תתקבל		useState<number>(0); let count:number = 0;
לשנות את הערך		setCount(count + 1); count = count + 1;

הבה נדגים בדרך אחרת. ניצור קומפוננטה אינטראקטיבית שבה יש מספר. המטרה שלנו היא להציג את מספר הפעם שהמשתמש עבר עם העכבר.

נשמע מפיח? לא ממש. ניצור אלמנט HTML פשוט, שלו נצמיד onMouseover שמאפיין פונקציה שבה המונה משתנה. ניצור את הקומפוננטה באופן רגיל ופשוט:

```

function ShowHover():JSX.Element {
  let time:number = 0;
  function countHover():void {
    time = time + 1;
  }
  return (
    <div>
      <h2 onMouseOver={countHover}>{time}</h2>
    </div>
  );
}
export default ShowHover;

```

אם תציבו את הקוד הזה ב-`ShowHover.tsx` ותקרוו לקומפוננטה מתוך `app`, תראו שלא משנה כמה פעמים אתם עוברים על הכפתור, המספר 0 יישאר על הלוח. למה? כי הקומפוננטה לא עברה רנדום מחדש. נכון, הקлик שינה את המשתנה הפנימי, אבל כל עוד אין רנדום חדש, אנו רואים את תוצאת הרנדום הקודם גם אם ה-`time` משתנה.

הפתרון הוא לחת לה את הזיכרון הזה לפי שלושת הצעדים שמנינו קודם:

1. לבצע `import` להוק הנכון, במקרה זה `useState`.
2. לומר לקומפוננטה בהתחלה איזה משתנה נכנס לזכרון (במקרה שלנו `time`) ואת הסוג שלו. להגדר פונקציה שתשנה את המשתנה בזיכרון. שם הפונקציה הוא תמיד `set` ואז שם המשתנה מתייחל באות גדולות.
3. לשנות את הקומפוננטה שצריך באמצעות הפונקציה שמשנה את המשתנה.

```
import { useState } from 'react';
function ShowHover():JSX.Element {
  const [time, setTime] = useState<number>(0);
  function countHobver():void {
    setTime(time + 1);
  }
  return (
    <div>
      <h2 onMouseOver={countHobver}>{time}</h2>
    </div>
  );
}
export default ShowHover;
```

זה כל מה שצריך לעשות.  
חשוב לציין כי לא כדאי להשתמש בסטייט בלי צורך. הסטייט וניהולו הם חלק חשוב מאוד בריект ואני נלמד בהמשך על דרכים שונות לניהול סטייט גלובלי. בנוסף על כן, נלמד גם על עוד הוקים.

**תרגיל:**

צרו קומפוננטה בשם CountDown.tsx שיש בה כפטור ומספר שמתחל ב-10. הקומפוננטה תספור המספר זהה לאחר (אין צורך לעצור ב-0).

**פתרונות:**

```
import { useState } from 'react';
function CountDown():JSX.Element {
  const [count, setCount] = useState<number>(10);
  function countUp():void {
    setCount(count - 1);
  }
  return (
    <div>
      <h2>{count}</h2>
      <button onClick={countUp}>Click me</button>
    </div>
  );
}
export default CountDown;
```

הקומפוננטה זו עשוה בדיקו אותו הדבר כמו הקומפוננטה של CountUp שיצרנו בפרק, אבל במקום להעלות את count ב-1 היא מורידה אותו ב-1. גם פה כדאי לשים לב לשלוות החלקים:

1. ה-import השונה (מייבאים את ההוק של useState).
2. קובעים את המשתנה שיכנס ל זיכרון של הסטיט ו את שם הפונקציה שמכניסה את המשתנה ל זיכרון. שם הפונקציה תמיד מורכב מ-set ומשם המשתנה. ניתן לקבוע את הערך ההתחלתי של הסטיט בשלב זהה. לא נשכח להגיד את סוג המידע של הסטיט.
3. כשרוצים לשנות את המשתנה, משתמשים תמיד בשם הפונקציה שמכניסה את המשתנה ל זיכרון.

**תרגיל:**

בкомпонנטה הקודמת שיצרתם, צרו קומפוננטה שמקבלת את המספר ההתחלתי ב-props בשם `time`. וכן הספירה תיפסק ב-0.

**פתרון:**

```
import { useState } from 'react';

type propsInfo = {
    time:number,
}

function CountDown(props:propsInfo):JSX.Element {
    const [count, setCount] = useState<number>(props.time);
    function countUp():void{
        if (count > 0) {
            setCount(count - 1);
        }
    }
    return (
        <div>
            <h2>{count}</h2>
            <button onClick={countUp}>Click me</button>
        </div>
    );
}
export default CountDown;
```

אני קורא לקומפוננטה זו כמוובן כה:

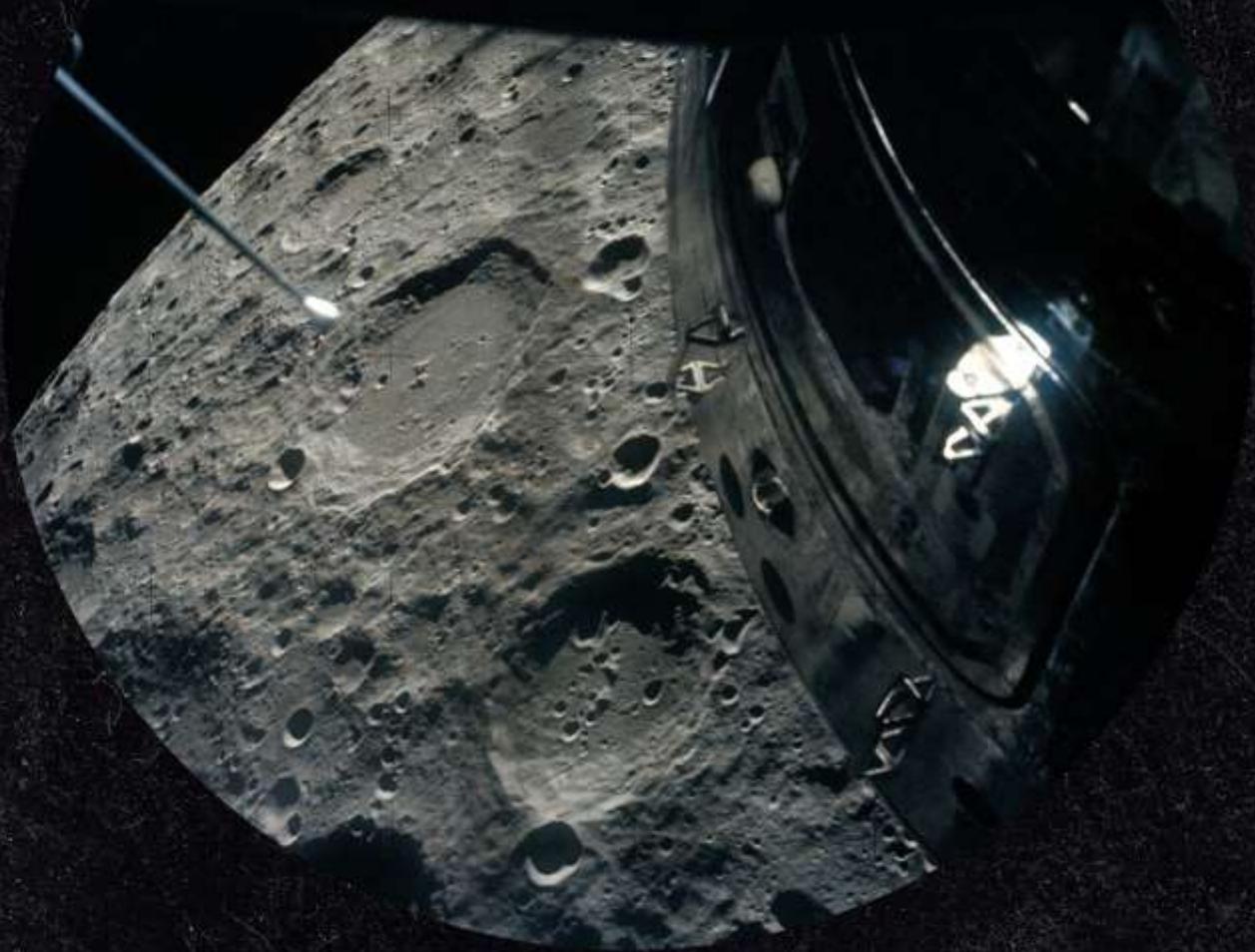
```
<CountDown time="3" />
```

הפתרון כמעט זהה לפתרון הקודם. כדאי לשים לב שאני לא יכול להעביר "3" כמחרוזת טקסט אלא מספר ופה אני נעזר ב-JSX ובסוגרים המ솔לים על מנת לעשות זאת. המטרה היא להבין

שהסתוייטים אינם תורה מסיני. הם פשוט מהו ששים בזיכרון, זה הכל, ואם לא משנים להם את הערך אפשר לגשת אליהם כרגע.  
צריך לזכור כמהן חשוב להגדיר את סוג המידע כמו שצרים.

פרק 10

# תכנון תבנה היקומפוננטות



# תכנון מבנה הקומפוננטות

אחד העקרונות החשובים של עבודה עם קומפוננטות הוא המונח "הרכבה" (composite), שמשמעותו הכנסה של קומפוננטות בתוך קומפוננטות. כל קומפוננטה היא יחידה עצמאית שמשתמשת בקומפוננטות מסוימת, ואפשר לחת את הקומפוננטות ולשים אותן בכל מקום אפשרי. כך למשל אם יש לי קומפוננטה המציגה טקסט בצורה נאה או מיוחדת, אני יכול להשתמש בה בהמון קומפוננטות אחרות. לקומפוננטה המציגה טקסט לא ממש אכפת מה היא מציגה כל עוד היא מקבלת את המידע ב-prop המתאים.

הבה נניח שיש לי קומפוננטה שהקלט שלה (כלומר מה שהיא מקבלת ב-props) הוא מילישניות וഫטט שלה הוא זמן מעוצב בצורה נאה וקראית. שם הקומפוננטה הוא Watch.tsx:

```
type propsInfo = {
  miliseconds: number;
};

type optionsInfo = {
  weekday: "long";
  hour: "numeric";
  minute: "numeric";
  second: "numeric";
};

function Watch(props: propsInfo): JSX.Element {
  const date = new Date(props.miliseconds);
  const options: optionsInfo = {
    weekday: "long",
    hour: "numeric",
    minute: "numeric",
    second: "numeric",
  };
  const time: string = date.toLocaleDateString("he-IL", options);
  return <span>{time}</span>;
}
export default Watch;
```

זו קומפוננטה אופיינית מאוד לקומפוננטת תצוגה בלבד. אנחנו משתמשים לשומר על הקומפוננטות שלנו קטנות ככל האפשר ולא להכניס לוגיקה ופונקציות מיותרות לקומפוננטת תצוגה. פה היא מקבלת מספר ומירה אותו לתצוגה בעברית. זה הכל.

אם-arצה להשתמש בה, אני אצור קומפוננטה אחרת שמעבירה אליה את הזמן שאנו רוצה שkomponentet הבודת תציג. למשל, קומפוננטת `TodayTime.tsx` שמעבירה את הזמן של היום אל

`komponentet Watch.tsx`

```
import Watch from './Watch.tsx';
function TodayTime():JSX.Element {
const today:number = Date.now();
return (
<Watch milliseconds={today} />
);
}
export default TodayTime;
```

את קומפוננטת `TodayTime.tsx` אני שם בכל מקום שבו אני צריך את השעה הנוכחית. אבל אם אני אציב את הקומפוננטה זו במקומות כלשהו – למשל `app`, הדף הראשי של אפליקציית הריאקט

- אני אראה מהו ממש מרגין:

```
import './App.css';
import TodayTime from './TodayTime';
function App():JSX.Element {
return (
<div className="App">
<header className="App-header">
<TodayTime />
</header>
</div>
);
```

```
}
```

```
export default App;
```

נסו את זה בעצמכם. זה קרייטי לתרגולים הבאים.  
למה השעון לא מתעדכן? אם קראתם את הפרק הקודם – וגם תרגלו אותו – אתם כבר יודעים שהרנדור של הקומפוננטה רץ פעם אחת בלבד וושאул מנת לעדכן אותו אנחנו צריכים לעשות משהו – כמו למשל `setInterval`. מה הבעיה? אני יכול לשים את ה-`setInterval` בתוך הקומפוננטה, אבל אני לא רוצה לשים אותו ב-`Watch`. מודיע?>User להסביר לנו מה קומפוננטת `Watch` זה לא רעיון טוב.

בתחילת הפרק כתבתי שחלק מתוכנו נכוון של מערכת עם קומפוננטות הוא תכנון קומפוננטות קטנות ככל האפשר. אבל מעבר לתיאוריה – אם אני אציב `setInterval` ב-`Watch`, אני לא אוכל להשתמש בה במקומות אחרים – כמו למשל במצב תצוגה של שעון קבוע (לדוגמא: השינוי האחרון באתר בוצע בשעה נס' וכך). מי צריך להעלות את הזמן בכל שנייה ולהציג את הזמן בסטייט `TodayTime` הינו קומפוננטה האב `xs`.

את זה ניתן לעשות בקלות עם `setInterval`:

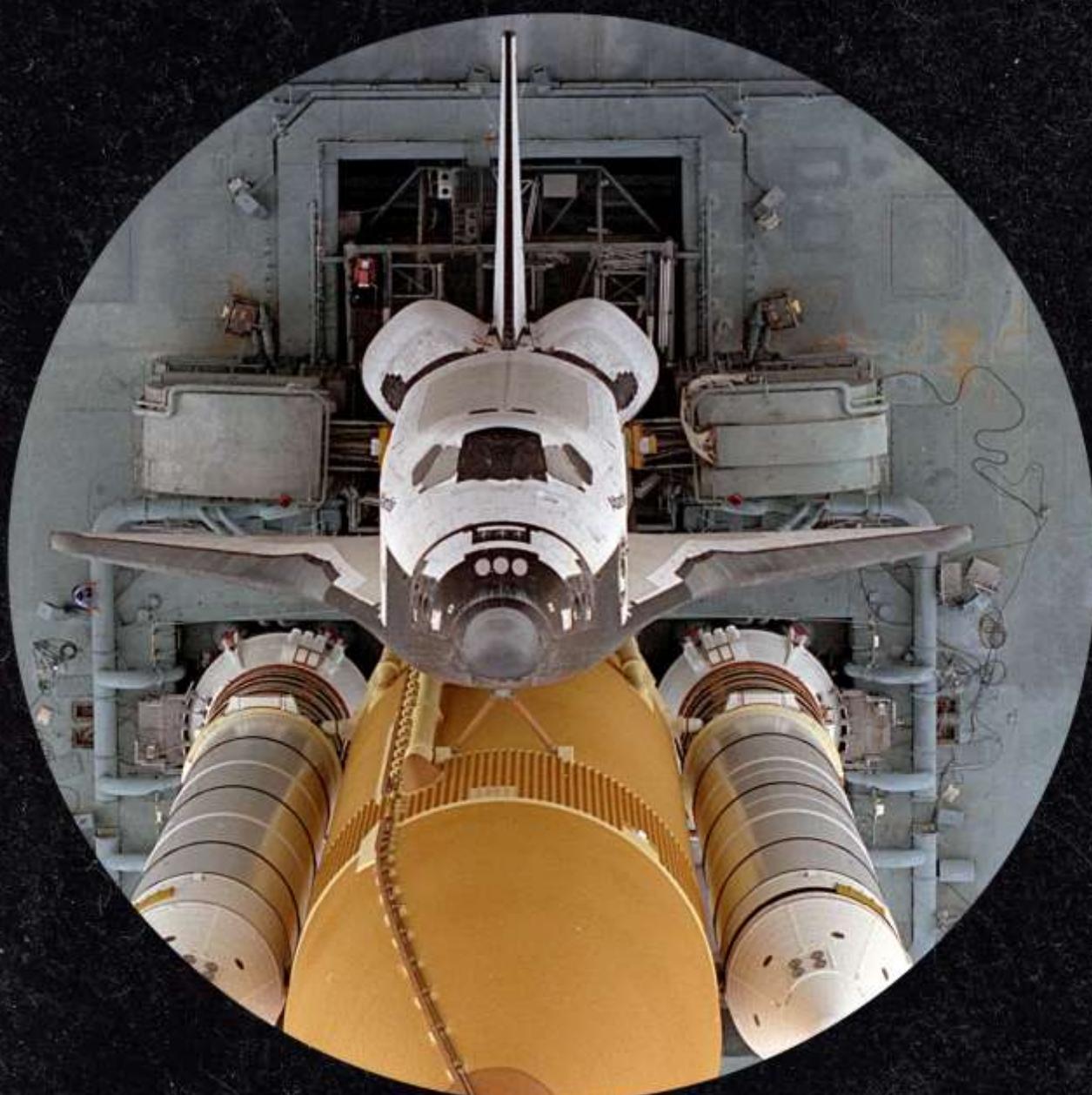
```
import { useState } from 'react';
import Watch from './Watch.tsx';

function TodayTime(): JSX.Element {
  const [today, setToday] = useState<number>(Date.now());
  function upTime(): void {
    setToday(Date.now());
  }
  setInterval(upTime, 1000);
  return (
    <Watch milliseconds={today} />
  );
}
export default TodayTime;
```

זה יעבוד מעולה. ככה עובדים בפיתוח קומפוננטות מודרני. קומפוננטה שמובילה לקומפוננטה  
שמובילה לקומפוננטה. עליינו לשאוף ככל האפשר לקומפוננטות קטנות בעלות תפקודיות מוגבלת  
שנוכל למחזר שוב ושוב. למשל, בכל אתר גדול או אפליקציה יש ספרייה פנימית של קומפוננטות  
תציגה שהמתכנתים ממחזרים שוב ושוב, וככה חוסכים זמן פיתוח. אבל כמו כל דבר בחיים,  
אם מגזינים במשהו זה עלול להוביל לביעות. כאשר אנו יוצרים יותר מדי קומפוננטות, אנו עלולים  
לייצור בעיות בייצועים וזמן הריצה ארוך. אף על פי שאין כלל אצבע מסודר במקרה הזה, צריך לזכור  
לא להגזרים.

פרק 11

# אידועים ועדכונים קומפוננטות



## אירועים ועדכון קומפוננטות

עד כה הצגנו קומפוננטות סטטיות או מקסימום כאלו שמתעדכנות כתוצאה מפעולות פנימיות כמו `setInterval`. אבל חלק גדול מהקומפוננטות אמורות לדעת לטפל בקלט מהמשתמש – קלומר פעולות ואירועים. אירועים הם חלק מהותי מג'אוوهסקרייפט ומג'אוوهסקרייפט שרך בסביבת דפדפן, וריאקט יודעת לעבוד איתם יפה מאוד. על מנת לדעת איך עובדים עם אירועים, נזכיר איך אירועי DOM, קלומר אירועים טبויים בג'אוوهסקרייפט, עובדים.

## AIROUT DOM

בגאומסתקראיפט, כאשר אנו עובדים עם אלמנטים טבעיים של HTML, כמו למשל `input`, אנו יכולים להציג להם אירועי האירועים הללו נקראים **AIROUT DOM**. להלן לוח של כמה אירועים נפוצים ולצדיהם דוגמאות:

דוגמה	שם אירוע
<code>&lt;button onclick="myFunction()"&gt;Click&lt;/button&gt;</code>	קליק
<code>&lt;select onchange="myFunction()"&gt;</code>	בחירה
<code>&lt;input type="text" onblur="myFunction()"&gt;</code>	יציאה מהאלמנט (יציאה מפוקוס)
<code>&lt;input type="text" onfocus="myFunction()"&gt;</code>	כניסה לאלמנט (פוקוס)

האירועים הללו זמינים לאלמנטים הטבעיים של HTML ואני מניח שאתם מכירים אותם. כדי לשימושם לב כל האירועים הללו כתובים באותיות קטנות ושאנו מעבירים אותם לאלמנט הטבעי בחרוזת טקסט שרצה ברגע שהאירוע מופעל.

אנו לא נוכל להשתמש בהם ב-XJS. אלו אירועי ב-HTML, ו-XJS שונה מ-HTML, למרות הדמיון הוויזואלי.

## אירועים סינטטיים/ ריאקטיביים

ב- JSX אנו יכולים להשתמש באירועים ריאקטיביים לאלמנטים שידועים לעובוד איתם. הם דומים לאירועי ה-DOM וגם מתנהגים כמוותם. מדובר בעצם באירועים עוטפים שיש להם אותו משק פעולה. הרשימה המלאה והארוכה מאוד של האירועים נמצאת בדוקומנטציה של ריאקט:

<https://reactjs.org/docs/events.html>

צריך לזכור שבעצם לכל אירוע DOM רגיל יש מקבילה ריאקטיבית. בדרך כלל המקבילה הריאקטיבית תהיה ב-case camel, כלומר האות הראשונה של המילה השנייה תהיה אות גדולה באנגלית.  
למשל:

אירוע ריאקט	אירוע DOM
onClick	onclick
onChange	onchange
onBlur	onblur
onFocus	onfocus

МОובן שאין לא מעביר לאירוע מחרוזת טקסט אלא ביטוי בתוך סוגרים מסולסלים על מנת ש-JSX ידע להתמודד איתו. מעבר זהה, השימוש באירועים זהה לחלווטין בריאקט. כך, למשל, אם יש לי קומפוננטה שנקראת MyInput ובה אלמנט `input` של HTML (מדובר באלמנט שמאפשר לנו להקליד טקסט בתוכו), אני יכול להשתמש ב-`on` כדי להציב את מה שהמשתמש כותב בסיטייט ומה שעשות אליו מה שבא לו.

נדגים עם קומפוננטה פשוטה מאד, שבה יש `input` שכל מה שנקליד בו יופיע על המסך שלנו. ניצור קומפוננטה שנקראת `tskInputViewer` ובה יהיה שני אלמנטים – `input`-`span` שיכיל את הטקסט. מה עם הטקסט? נשים אותו ב-`state` בבדיקה כפי שלמדנו.

```

import { useState } from 'react';

function InputViewer(): JSX.Element{
  const [text, setText] = useState<string>('');

  return (
    <div>
      <span>{text}</span>
      <input type="text" />
    </div>
  );
}

export default InputViewer;

```

איך אנחנו מعتبرים את המידע מתוך ה-`input` אל הסטיטו `text`? בדיק בשביל זה אנחנו משתמשים ב-`onChange`, שלפי שמו אנו רואים שהוא אירוע ריאקטיבי. איך הוא עובד? באמצעות תכונה שמצוידים לאלמנט `onClick`. התכונה זו מקבלת פונקציה שמופעלת כאשר האירוע מופעל. למשל:

```
<input onClick={e=> console.log(e)} type="text" />
```

האירוע הוא `onClick` שמתקיים כאשר אנו מקליקים על האלמנט. כאשר אנו מקליקים, הפונקציה שיש בתוך התכונה `onClick` מופעלת. כמו כן הפונקציה זו:

```
e => console.log(e)
```

זו פונקציית חץ שפשות מופעלת עם `e`. מה זה? האירוע עצמו, כלומר אובייקט עם מידע על האירוע. יש בו אוסקריפט כמה אירועים שאפשר להציג לאלמנטים שונים.

הקוד הזה למשל:

```
<input onMouseEnter={e=> console.log(e)} type="text" />
```

הוא אירוע `onMouseEnter` שמתקיים כאשר המשתמש עובר עמו העכבר מעל האלמנט המذובב, במקרה זהה `input`, שדה הטקסט. אם זה קורה, הפונקציה:

```
e=> console.log(e)
```

מופעלת. לא צריך להיבהל מהסינטקס המוזר. מדובר בפונקציית חץ שהיא חלק מג'אוועסקריפט. פונקציית החץ זו מקבלת אירוע `e` ומדפיסה אותו בקונסולה. אני ממליץ לכם לנסות אותה בקומפוננטה שיש לעיל או בכל קומפוננטה. האירוע הזה יעבד גם על כל אלמנט אחר.

אנחנו לא חייבים להשתמש בפונקציית חץ. אפשר גם להגיד בקומפוננטה שלנו פונקציה ולהעביר את השם שלה ב-`JSX` שלנו. למשל:

```
import { useState } from 'react';
function InputViewer():JSX.Element {
  const [text, setText] = useState<string>('');
  function clickHandler(e:
    React.FormEvent<HTMLInputElement>):void {
    console.log(e);
  };
  return (
    <div>
      <span>{text}</span>
      <input onClick={clickHandler} type="text" />
    </div>
  );
}
export default InputViewer;
```

זו דרך מקובלת יותר לעבוד ואולי זה יהיה פחות מפחיד מפונקציית חץ. נסו את הקוד הזה עכשוון.  
הקליקו על ה-`ה-click` ותראו איך פונקציית `clickHandler` מופעלת בכלל לחיצה. כדאי לשים לב שפה  
אני מחויב להגדיר גם את הקלט וגם את הפלט.

### `React.FormEvent<HTMLInputElement>():void`

הקלט של הפונקציה הוא אירוע. שם האירוע הוא `React.FormEvent` ואני נדרש להגדיר מה בדיק  
אני מעביר, במקרה הזה `HTMLInputElement` והפונקציה לא מחזירה כלום אז אני קובע `void`: -  
זה נראה קצת מפחיד אבל מתכונתי ריאקט מנוסים מוסיף את האירועים האלה כבר באופן חצי  
אוטומטי וסוג המידע הזה נמצא כמעט כמעט בכל אירוע

יש ברגע זה סקריפט לא מעט אירועים - חלק מהם, כאמור, עובדים על אלמנטים מסוימים וחלק  
לא. האירועים האלה עובדים על אלמנט HTML בלבד, כמובן, לא על אלמנטים ריאקטיים. אבל  
כרגע אין לנו בעיה, יש לנו אלמנט HTML פשוט. איך אנחנו מעבירים את מה שאנו מקלידים בתוך  
ה-`ה-input` אל הסטייט?

אנו נשתמש באירוע הריאקטי `shnkrat` `onChange` וশמו פועל בכל פעם שיש שינוי בערך אלמנט  
ה-`ה-input`. האירוע יפעיל פונקציה שנקראת `changeHandler` והוא תבצעঅস্লোস `shl` הסטייט  
באמצעות `setText` שאותו הגדרנו באמצעות הוק.

```
import { useState } from 'react';

function InputViewer():JSX.Element {
  const [text, setText] = useState<string>('');
  function changeHandler(e: React.FormEvent<HTMLInputElement>):void {
    setText(e.currentTarget.value);
  }
  return (
    <div>
      <span>{text}</span>
      <input onChange={changeHandler} type="text" />
    </div>
  );
}
```

```

        </div>
    );
}

export default InputViewer;

```

צרו את הקומפוננטה הזו והקלידו בשדה `theInput`. תוכלו לראות שהtekst מופיע מיד ב-`span`.  
מדובר אנחנו משבירים אל הסטייט רק את `e.target.value`? כי זה החלק באובייקט האירוע,  
שנאמר תכונת האירוע מעבירה לנו, שמכיל את המידע שמעוניין אותנו. באובייקט האירוע יש  
מידע רב ואתם מוזמנים לבחון אותו עם הקונסולה או עם הדיבאגר, אבל ב-`onChange`, המידע  
הтекסטואלי שיש בשדה מועבר עם `e.target.value`.

אנחנו יכולים למשך כל התנהגות שאנו רוצים באמצעות אירוע. למשל, נניח שאנו רוצים שהמידע  
מתוך שדה `theInput` יופיע בתחום `theSpan` רק כאשר אנו מקליקים על כפתור. איך נעשה זהה דבר?  
ראשית, ניצור כפתור HTML פשוט שייהי לו אירוע `onClick`.

```
<button onClick={clickHandler}>Click me</button>
```

עכשו ניצור את הפונקציה ש יודעת לעבוד עם האירוע. במקרה הזה מדובר באירוע קליק שלא  
מכיל מידע, אז איך אני אדע לבדוק מה יש בתחום `theSpan`? טוב, זה קל – יש לי את אירוע  
ה-`changeHandler` שמכניס כל מה שאני מקליד ישירות בשדה `theInput` אל משתנה סטייט  
שנקרא `text`. אני צריך רק לנתק את `theSpan` מה-`text` ולה לחבר אותו למשנה סטייט אחר (אקרא  
לו `viewText`), והלחיצה תגרום לסטייט `viewText` להתאכלה מסטייט `text`.

```

import { useState } from 'react';

function InputViewer():JSX.Element {
  const [text, setText] = useState<string>('');
  const [viewText, setViewText] = useState<string>('');

  function changeHandler(e
  :React.ChangeEvent<HTMLInputElement>):void {
    setText(e.target.value);
  }

  function clickHandler():void {
    setViewText(text);
  }

  return (
    <div>
      <span>{viewText}</span>
      <input onChange={changeHandler} type="text" />
      <button onClick={clickHandler}>Click me</button>
    </div>
  );
}

export default InputViewer;

```

העניין פה הוא להבין שבסופו של דבר, האירועים בריאקט נראים למשתמש באופן כמעט זהה לאירועי ה-DOM הטבעיים ולא צריך להיבהל מהם. אנו ממצידים את האירועים הריאקטיים לאלמנטי HTML שתומכים בהם כמעט כמו האירועים של ה-DOM, אך בדומה הבדלים קלים - שם הפקנציה שונה (checkbox בעטיפה הריאקטית, onclick באירוע הטבעי) וכמובן מה שהוא מעבירים (פונקציה JSX שעוטופה ב-{ }) בריאקט ומחזקת טקסט באירוע הטבעי). אם היינו משתמשים בג'אוויסקcript רגיל, הפעלת אירוע הייתה נראה כך:

```
<button onclick="clickHandler()">Click me</button>
```

אבל בغال ה- JSX, אנו עושים את זה ללא הפעלה ובלוי סוגרים מסולסים. יש עוד כמה שינויים קלים בין אירוע טבעי לאיורע ריאקטיבי. באירוע קליק ריאקטיבי, למשל, אנו יכולים את הפעוף לא באמצעות return false אלא באמצעות שימוש בפונקציה stopPropagation, אך לא נדון בכך בפרק זה.

אבל איך אנו מצדדים אירועים לקומפוננטה ריאקטיבית? פה העניינים מסתבכים מעט, אבל ממש מעט. קומפוננטה ריאקטיבית יכולה להיות למשל קומפוננטה שמכילה `input`. מהו זה:

```
function Input():JSX.Element {
  return (
    <input type="text" />
  );
}

export default Input;
```

זה בטח נראה לכם מוגוחך, אבל זה לא מאד מוגוחך. מקובל מאוד לעתוף כל אלמנט HTML בריект, במיוחד שדות קלט, בקומפוננטה עצמאית. אנו עושים את זה פעמים רבות כי בקומפוננטות המכילות עיצוב, יש גם עיצוב מיוחד בקומפוננטה שימושה על ה-`input` או על ה-`button`, אז זה לא כל כך מופר.

אם אני רוצה להשתמש בקומפוננטה `asInput` בדוגמה שהבאתי קודם לכן, זה לכארה פשוט – אני רק אחליף את האלמנט `input` בקומפוננטה `asInput`. מהו בסגנון זהה:

```
import { useState } from 'react';
import Input from './Input';

function InputViewer():JSX.Element {
  const [text, setText] = useState<string>('');

  function changeHandler(e
:React.ChangeEvent<HTMLInputElement>):void
  {
    setText(e.target.value);
  }

  return (
    <div>
      <span>{text}</span>
      <Input onChange={changeHandler} type="text" />
    </div>
  );
}

export default InputViewer;
```

אבל אם לא תטעלו ותנסו את הדוגמה בעצמכם, ככלומר תעתייקו את `Input.tsx` לתוך הפרויקט, Vite, תיצרו את קומפוננטת `InputViewer.tsx` ותציבו אותה באפליקציית הריאקט שלכם ב-קמארה, תראו שהוא לא עובד. למה? כי קומפוננטות ריאקט לא מקבלות אירוחים באופן טבעי.

از מה עושים? פשוט מאד. בקומפוננטה שלנו אנו דואגים לקבל את הפונקציה שמטפלת באירוע מה-props, בדוק כמו כל משתנה או נתון אחר, ואז את מה שאנו מקבלים מה-props אנו מעבירים לאלמנט ה-HTML הטבעי.

```
type propsInfo = {
  onChange: (e:React.ChangeEvent<HTMLInputElement>) => void,
};

function Input(props:propsInfo):JSX.Element {
  const changeHandler = props.onChange;
  return (
    <input onChange={changeHandler} type="text" />
  );
}

export default Input;
```

מה בעצם מתרחש כאן?

בקומפוננטה שמשתמש ב-`Input` אני לוקח את ה-`props.onChange` ומעביר אותו לאלמנט ה-`input` הטבעי. כך, אם אני משתמש בקומפוננטה `Input` באופן הבא:

```
<input onChange={changeHandler} type="text" />
```

היא תעבור לי.

כיוון שאנחנו עובדים בטיפס קרייפט, אני מגדיר את סוג המידע ש-`onChange` מחזיר כ-`void`.

בואו נדגים שוב עם הדוגמא אחת הדוגמאות הקודמות. אני אזכיר קומפוננטת כפטור ריאקטיבית שעוטפת כפטור HTML רגיל. שוב, זה נשמע מאולץ, אבל זו פרקטיקה נפוצה מאוד בספריות ובאפליקציות, ואם יצא לכם לעבוד או לראות קוד אמיתי בריאקט, תוכלו לראות את זה.

קומponent Ax Button.tsx תיראה כך:

```
type propsInfo = {  
  onClick: (e: React.MouseEvent<HTMLButtonElement>) => void;  
};  
  
function Button(props: propsInfo): JSX.Element {  
  
  const clickHandler = props.onClick;  
  
  return (  
    <button onClick={clickHandler}>Click</button>  
  );  
}  
  
export default Button;
```

גם כאן, אני לוקח את כל מה שמעבירים לקומפוננטה ב-`props` ו מעביר את זה להאה, לאלמנט `button` הרגיל, של ה-HTML, שיודע לעבוד עם אירועים. והשימוש? כרגע:

```
import { useState } from 'react';
import Input from './Input';
import Button from './Button';

function InputViewer():JSX.Element {
  const [text, setText] = useState<string>('');
  const [viewText, setViewText] = useState<string>('');
  function changeHandler(e:React.ChangeEvent<HTMLInputElement>):void {
    setText(e.target.value);
  };
  function clickHandler(e:React.FormEvent<HTMLInputElement>):void {
    setViewText(text);
  };
  return (
    <div>
      <span>{viewText}</span>
      <Input onChange={changeHandler} />
      <Button onClick={clickHandler} />
    </div>
  );
}

export default InputViewer;
```

זה עלול להיות קצת מבלבל, במיוחד בגלל סוג המידע המפচידים, אבל כדאי לזכור שמדובר בסוףו של יום ב-`useState` שכבר Learnedנו לעבוד איתם יפה מאוד ושאותם אנו מעבירים לקומפוננטה בקלהות.

```

13  function clickHandler(e) {
14    setViewText(text);
15  };
16
17  return (
18    <div>
19      <span>{viewText}</span>
20      <Input onChange={changeHandler} type="text" />
21      <Button onClick={clickHandler}>Click me</Button>
22    </div>
23  );
24}
25
26
@ Button.jsx ●
src > @ Button.jsx > ...
1 import React from 'react';
2
3 function Button(props) {
4
5   const clickHandler = props.onClick;
6
7   return (
8     <button onClick={clickHandler}>Click</button>
9   );
10 }
11
12 export default Button;

```

אם לאלמנט הטבעי ב-HTML יש את כל האירועים שבאים במתנה, בקומפוננטה ריאקטית אנחנו חייבים להתייחס לכל אירוע שהוא רצוי ב-props ולהעביר אותו אל האלמנט הטבעי.

**תרגיל:**

צרו קומפוננטה בשם MyDivContainer.tsx שמכילה div שיש בתוכו טקסט "My Div". כאשר העכבר עובר מעל הטקסט, תופיע המילה active באוטו div.

**רמז:** יש צורך להציג שני מנהלי אירועים: הראשון הוא onMouseOver, שמכניס לסתיט משטנה כלשהו ואת המילה active, והשני הוא onMouseOut, שמאפס את המשטנה.

**פתרונות:**

```
import { useState } from 'react';

function MyDivContainer(): JSX.Element {
  const [activeText, setActiveText] = useState<string>('');

  function mouseoverHandler(): void {
    setActiveText('active');
  }

  function mouseoutHandler(): void {
    setActiveText('');
  }

  return (
    <div
      onMouseOver={mouseoverHandler}
      onMouseOut={mouseoutHandler}>
      MyDiv {activeText}
    </div>
  );
}

export default MyDivContainer;
```

על אותו אלמנט יש לנו (שיםו לב שאלמנט זה הוא אלמנט HTML טبוי שיכול לקבל אירועים שניים שונים. האחד, `MouseOver`, נכנס לפעולה כאשר העכבר נמצא מעל האלמנט, והשני, `MouseOut`, נכנס לפעולה כאשר העכבר יוצא ממנו. כל מה שנותר לי לעשות הוא ליצור את הפונקציות שפועלות בזמן שהאירוע מתרחש. הראשונה לוקחת סטייט שיצרת ומכניסה לתוכו את המילה `active` והשנייה מאפסת אותו. אני מכניס את המשתנה `shish` בסטייט ועובד איתנו.

**שימוש לב:** אפקט מעבר עושים בדרך כלל עם CSS פשוט. כאן אנו עושים את זה עם ריאקט לשם התרגול. בנוסף על כן, `mouseOver` הוא גם אירוע עיתוי שיכול ליצור זליגות זיכרון.

**תרגיל:**

ארכיטקט המערך הביט בתרגיל הקודם וביקש שה-`Div` יהיה קומponent ריאקט. ככלומר שהקומponentה הקודמת תיראה כך:

```
import { useState } from 'react';
import Div from './Div';

function MyDivContainer():JSX.Element {
  const [activeText, setActiveText] = useState<string>('');
  function mouseoverHandler():void {
    setActiveText('active');
  }
  function mouseoutHandler():void {
    setActiveText('');
  }
  return (
    <div>
      <Div
        onMouseOver={mouseoverHandler}
        onMouseOut={mouseoutHandler}>
      </Div>
      {activeText}
    </div>
  );
}

export default MyDivContainer;
```

צרו את קומponent `Div` ב-`xs.Div` כדי שהקוד שלעיל יעבד.

**רמז:** onMouseOut ו onMouseOver מועברים לקומפוננטת Div כ-props ועליכם להעביר אותם אל ה-div הטבעי של ה-HTML בקומפוננטת Div.

**פתרונות:**

```
import { useState } from 'react';
type propsInfo = {
  onMouseOver: () => void;
  onMouseOut: () => void;
}
function Div(props:propsInfo):JSX.Element {
  let onMouseOver:()=>void = props.onMouseOver;
  let onMouseOut:()=>void = props.onMouseOut;
  return (
    <div
      onMouseOver={onMouseOver}
      onMouseOut={onMouseOut}>
      My div
    </div>
  );
}
export default Div;
```

אפשר גם לkür את התהיליך ולכתוב משהו כמו:

```
import { useState } from 'react';

type propsInfo = {
  onMouseOver: () => void;
  onMouseOut: () => void;
}

function Div(props: propsInfo): JSX.Element {
  return (
    <div
      onMouseOver={props.onMouseOver}
      onMouseOut={props.onMouseOut}
    >
      My div
    </div>
  );
}

export default Div;
```

כאמור, אם יש לי קומפוננטה ואני רוצה לקבל מקומפוננטת אב את האירועים, אני צריך לדאוג להעביר אותם בעצמי. הקומפוננטות של ריאקט לא מקבלות אירועים כמו האלמנטים הטבעיים. מה שאני עושה בקומפוננטת `Div` הוא לקבל את האירועים דרך ה-`props`, להעביר אותם למשתנים ולהציב אותם באירועים של האלמנט הטבעי:

```

16   return (
17     <div>
18       <Div
19         onMouseOver={mouseoverHandler}
20         onMouseOut={mouseoutHandler}
21       >
22       </Div>
23       {activeText}
24     </div>
25   );
26 }
27

```

**MyDivContainer.jsx**

---

```

@ Div.jsx •
src > @ Div.jsx > ...
1 import React, { useState } from 'react';
2
3 function Div(props) {
4
5   return (
6     <div
7       onMouseOver={props.onMouseOver}
8       onMouseOut={props.onMouseOut}
9     >
10    My div
11  </div>
12);
13

```

**Div.jsx**



בצם בדרך זו הfonקציות המטפלות באירועים עוברות מאלמנט האב אל ה-`props` בקומפוננטה שלוי, וברגע שהן ב-`props` אני יכול לשים אותן באלמנט שיש בקומפוננטה הפונה לה.

**תרגיל:**

צרו קומפוננטת Counter המציגת את הסירה 0 עם שלושה כפתורים. הראשון מעלה את המספר ב-1, השני מוריד את המספר ב-1 והשלישי מאפס את המספר.

**פתרונות:**

```
import { useState } from 'react';

function Counter(): JSX.Element {
  const [count, setCount] = useState<number>(0);

  function increaseHandler(): void {
    setCount(count + 1);
  }

  function decreaseHandler(): void {
    setCount(count - 1);
  }

  function restartHandler(): void {
    setCount(0);
  }

  return (
    <div>
      <button onClick={increaseHandler}>Increase</button>
      <button onClick={decreaseHandler}>Decrease</button>
      <button onClick={restartHandler}>Restart</button>
      <div>{count}</div>
    </div>
  );
}

export default Counter;
```

כאן אני מצמיד אירוע לכל אחד מהכפטורים שמשנה את הסטייט של count. כיוון שמדובר באלמנט button טבעי, אין לי שום בעיה להציג אליו אירוע פשוט כמו בקוד ג'אווהסקריפט רגיל.

פרק 12

# אלמנט FRAGMENT



# אלמנט **fragment**

כידוע, תמיד צריך להיות אלמנט אב אחד בקומפוננטת ריאקט. אם תנסה להחזיר בקומפוננטה משהו כמו:

```
return (
  <div>
    Foo
  </div>
  <div>
    Bar
  </div>
);
```

תקבלו הודעת שגיאה:

Parsing error: Adjacent JSX elements must be wrapped in an enclosing ?</>...<> tag. Did you want a JSX fragment

זה לא נראה כמו בעיה גדולה, נכון? כיון שתמיד אפשר לשימוש Div או span כABA. אבל זה כן בעייתי כי במקרה רבים אני לא רוצה שהיא לי אלמנט אב. למשל, אם אני יוצר קומפוננטות שמחלייפות זו, שזה אלמנט HTML שעוטף את השורה בטבלה. נניח משהו כמו:

```
function TableRow():JSX.Element {
  return (
    <tr><td>Foo</td></tr>
    <tr><td>bar</td></tr>
  );
}

export default TableRow;
```

אני אהיה בבעיה. אני לא יכול לעתוף אותו ב-`div` או `span` כי אם אני אציב אותו בתוך `Table` אני אקבל את המבנה הזה:

```
<table>
  <div>
    <tr><td>Foo</td></tr>
    <tr><td>bar</td></tr>
  </div>
</table>
```

זה מבנה לא תקין. המונ פעים אני גם לא רוצה להכניס `div` או `span` כי אני לא יודע בתוך איזו קומפוננטה תוצב הקומפוננטה שלי וזה עלול להיות בעייתי. כמה טוב היה אם היינו יכולים להכניס אלמנט אב "ש��וף"! אז כן, יש אלמנט כזה שנקרא `React.Fragment`. האלמנט זהה יכול להיות במקום אלמנט אב והוא לא יודפס כלל. החל מהדוגמאות הבאות אני משתמש בו.

```
function TableRow(): JSX.Element {
  return (
    <React.Fragment>
      <tr><td>Foo</td></tr>
      <tr><td>bar</td></tr>
    </React.Fragment>
  );
}

export default TableRow;
```

השם שלו מעט מרתייע ונראה מפחיד, אבל זה פשוט מאד – מדובר באלמנט אב ש��וף, כזה שלא מודפס ולא מתייחסים אליו. מקובל מאוד להשתמש בו בקומפוננטות בסיס ובקומפוננטות תצוגה.

מקובל גם להשתמש בקיצור `<1>`.

```
function TableRow():JSX.Element {  
  
  return (  
    <>  
      <tr><td>Foo</td></tr>  
      <tr><td>bar</td></tr>  
    </>  
  );  
}  
  
export default TableRow;
```

פרק 13

# USEFFECTS



# useEffects

כפי שלמדנו בפרק על הסטייט, כשהקומפוננטה מסויימת משתנה, למשל בגלל אירוע מסוים, היא מבנית מחדש על התצוגה שלה. הבנייה זו מוחדש נקרואת `רנדור`. הבה נציג באמצעות קומפוננטה שנבנתה באחד התרגילים בפרק על אירועים – הקומפוננטה `Counter.tsx`:

```
import { useState } from 'react';

function Counter(): JSX.Element {
  const [count, setCount] = useState<number>(0);

  function increaseHandler(): void {
    setCount(count + 1);
  }

  function decreaseHandler(): void {
    setCount(count - 1);
  }

  function restartHandler(): void {
    setTime(0);
  }

  return (
    <div>
      <button onClick={increaseHandler}>Increase</button>
      <button onClick={decreaseHandler}>Decrease</button>
      <button onClick={restartHandler}>Restart</button>
      <div>{count}</div>
    </div>
  );
}

export default Counter;
```

בפעם הראשונה, ובכל פעם שנעשה שינוי שמשפיע על התצוגה של הקומפוננטה זו (במקרה זה עם כפתורים), אבל זה יכול להיות שינוי שנוצר כתוצאה מ-`props` שהשתנו), מתבצע רנדור מחדש. לא מעט פעמים אנו רוצים לדעת שהוא רנדור או רנדור מחדש של הקומפוננטה. בדיק בשביל זה אנו יכולים ליצור הוק מיוחד שיפעל פונקציה שאנו מעבירים לו.

ההוק הזה נקרא **useEffect** ומשתמשים בו בדומה להוק של **useState**. הוא רק מרים פונקציה שאנו מעבירים לו אחרי כל רנדור.

הדרך הטובה ביותר להבין את זה היא באמצעות דוגמה:

```
import { useState, useEffect } from 'react';
function Counter():JSX.Element {
  const [count, setCount] = useState<number>(0);
  useEffect(()> void => { console.log('re-rendered!'); });
  function increaseHandler():void {
    setCount(count + 1);
  }
  function decreaseHandler():void {
    setCount(count - 1);
  }
  function restartHandler():void {
    setCount(0);
  }
  return (
    <div>
      <button onClick={increaseHandler}>Increase</button>
      <button onClick={decreaseHandler}>Decrease</button>
      <button onClick={restartHandler}>Restart</button>
      <div>{count}</div>
    </div>
  );
}
export default Counter;
```

ראשית הבאתי את useEffect כhook ב-import. הכי קל ונחמד בעולם. השלב הבא הוא פשוט להשתמש בהוק הזה. אני קורא ל-`useEffect` ו מעביר לו ארגומנט שהוא הפונקציה שתפעלה. הפונקציה המופעלת לא מחזירה כלום אז אני מגדיר את הפלט שלה כ-`void` כמו שצראיך בטיפוסкриיפט.

```
useEffect(()> void => { console.log('re-rendered!'); });
```

מה שיופיע הוא:

```
(()> void => { console.log('re-rendered!'); })
```

בכל פעם שהקומפוננטה תרוץ. נסו את זה! השתמשו בקומפוננטה זו והציצו בكونסולה של כל המפתחים כאשר אתם משנים את הערכים השונים. תראו שבעל פעם שיש שינוי בתצוגה של הקומפוננטה, הקונסולה מייצרת אירוע.

בחרתי בפונקציה חז' פשוטה, כמובן, להשתמש בשם של פונקציה, שהוא בסגנון זה:

```
function logEffect():void {
  console.log('re-rendered!');
}

useEffect(logEffect);
```

פונקציות חז' עלולות לבלבול, אבל צריך לזכור שהן בסופו של דבר פונקציות רגילות לכל דבר עם כמה תוספות חביבות, כמו למשל שמירה על `this`. זו הסיבה שכדי מאד להשתמש בהן.

אנו משתמשים ב-`useEffect` למגוון מטרות: בדרך כלל על מנת לקרוא ל-`API` חיצוני, ללוגים ולפעולות נוספות. תלוי במערכת. `useEffect` החליפה את "מעגל החיים הריאקטיבי" שהוא מקובל כשהשתמשו בקומפוננטות מבוססות קלאס. נרჩיב על כך בפרק על קומפוננטות מבוססות קלאס. `useEffect` מכיל פרמטר נוסף שומולץ להשתמש בו. הפרמטר הזה רלוונטי כאשר אנו רוצים למדוד רנדור חדש בעקבות שינויים `props` (ולא שינויים סטטיים). זהו מערך של התוכנות שאחריהן יש לעקוב וראם אם הן משתנות, הפונקציה ב-`useEffect` תרוץ.

כн, למשל, אם יש לי קומפוננטה מסוימת ואני רוצה להזכיר לרנדור מחדש שלה שמתבצע רק ברגע שינוי props מסוימים, אני אוסיף כארוגומנט שני את ה-`deps` הרלוונטיים:

```
import { useEffect } from 'react';

type propsInfo = {
  count: number;
}

export default function CountViewer(props:propsInfo):JSX.Element {
  const count:number = props.count;
  useEffect(()():void => console.log('Only props.count were re
rendered!', [props.count])
  return <div>{count}</div>
}
```

הדוגמה מדברת בעד עצמה. אם אנו מפרטים ארגומנט שני ל-`useEffect`, אז הפונקציה שהעבירהנו כארוגומנט הראשון תרוץ בהתאם לפרמטרים המפורטים בארגומנט השני.

פרק 14

# הומפוננטה ללא שם



# קומפוננטה ללא שם

עד כה השתמשנו בקומפוננטות בעלות שם, ככלומר יצרנו פונקציה בעלת שם ואז ייצאנו את הפונקציה החוצה באמצעות `export`. למשל קומפוננטה כזו:

```
type propsInfo = {
  onClick: ()=>void;
}

function Button(props:propsInfo):JSX.Element {
  const clickHandler = props.onClick;
  return (
    <button onClick={clickHandler}>Click</button>
  );
}

export default Button;
```

קרנו לפונקציה `Button` ואז ייצאנו אותה. זה סינטקס ולידי שnoch להבנה לאנשים שלומדים ריאקט לראשונה. אבל יש דרך נוספת לכתוב פונקציות. בסופו של דבר, למי שמייבא את הפונקציות ומשתמש בהן אין שום צורך בשם הפנימי הזה והוא רק לצורך הנוחות שלנו. אבל גם אנחנו לא זקוקים לשם זהה. ה-`export` מקבל גם פונקציות חז' אונומיות. ככלומר כל קומפוננטה שכתבנו עד כה תעבור באופן מושלם גם אם נמיר אותה לפונקציית חז' אונומית וניצא אותה. למשל, הפונקציה `Button` בהחלט תעבור כרגע אם היא תיראה כך:

```
type propsInfo = {  
  onClick: ()=>void;  
}  
  
export default (props: propsInfo):JSX.Element => {  
  
  const clickHandler = props.onClick;  
  
  return (  
    <button onClick={clickHandler}>Click</button>  
  );  
}
```

זה נראה מפחד, מבעית ואפילו מזרר לרוב המתכנתים שלא מרגלים בריקט. אבל אם הגעתם לפוך זהה וקראתם היטב את כל הפרקים - גם תרגלם - תוכלו להבין את הקוד הזה אם תיתקלו בו. מה שנעשה כאן הוא פשוט: במקום להציג על שם פונקציה, אני יוצר אותה כפונקציה אונומית ופשוט מחזיר אותה עם `export default`. הבעה המרכזית היא שכלי המפתחים של ריאקט לא יציג את שם הקומפוננטה ויקשה עליינו למצוא אותה.

פרק 15

# עליזוב קומפוננטות



## עיצוב קומפוננטות

יש כמה דרכים לעצב קומפוננטות בריект, חלון גם בעזרת ספריות עזר. בפרק זה אנו נתמקד בשתי דרכים עיקריות שבאות ייחד עם ריאקט. ריאקט עובדת עם CSS, שהוא סינטקס המאפשר לנו לעצב אלמנטים של HTML שימושיים בו בדף אינטרנט עם או בלי קשר לריקט. הספר הזה לא מלמד CSS בלבד מבוא קצר, שאמור להסביר לעיצוב בסיסי.

בדף אינטרנט בסיסים יש לנו קובצי CSS שאנו מיבאים לדף ה-HTML, אבל באפליקציה ריאקט אנו יכולים להשתמש ב-import, וספרייה וובפאק תdag לטען את הקובץ יחד עם הקומפוננטה.

ה-import כאן מבוצע בפשטות יבוא לשם הקובץ. בקובץ הראשי של אפליקציית הריאקט, App.js, אנו יכולים לראות את ה-import הזה:

```
import './App.css';
```

הוא אומר: "אני טוען את CSS זהה עם הקומפוננטה".  
קובץ CSS הוא קובץ פשוט מאוד, טקסטואלי, עם סימנת css, שמכיל את העיצוב של הקומפוננטה. מקובל לא להכניס עיצובים נוספים לקובץ זהה. הוא נטען אך ורק עם הקומפוננטה.

כדי להבין איך CSS עובד ואיך הוא עובד בפרויקט, אנו נעבד עם הקומפוננטה Counter.tsx שבנונו בפרק על אירופים.

```
import { useState } from 'react';

function Counter():JSX.Element {
    const [count, setCount] = useState<number>(0);

    function increaseHandler():void {
        setCount(count + 1);
    }

    function decreaseHandler():void {
        setCount(count - 1);
    }

    function restartHandler():void {
        setCount(0);
    }

    return (
        <div>
            <button onClick={increaseHandler}>Increase</button>
            <button onClick={decreaseHandler}>Decrease</button>
            <button onClick={restartHandler}>Restart</button>
            <div>{count}</div>
        </div>
    );
}

export default Counter;
```

ראשית, ניצור קובץ CSS שנקרא Counter.css ואז ניבא אותו. אנו נשים בראש הקומפוננטה את הטקסט:

```
import './Counter.css';
```

את הקומפוננטה נציג, כרגע, ב-App, הקובץ הראשי של אפליקציית הריאקט שלנו. מהרגע זהה, כל שינוי שנעשה ב-CSS יוצע בקומפוננטה. על מנת לבדוק שהכל עובד, נציג בקובץ Counter.css את הטקסט הזה:

```
button {  
  background-color: red;  
}
```

נסתכל בקומפוננטה ונראה שככל הרקע של הcptories הפך לאדום. חשוב מאד לציין: כיוון ש-class היא מילה שמורה בג'אווהסקריפט, JSX הוא ג'אווהסקריפט, אנו חייבים להשתמש ב-className ולא ב-class.

אם אתם יודעים CSS, עכשו אתם כבר יודעים איך לעצב קומפוננטה. לכל קומפוננטה-Amor להיות צמוד ה-CSS שלה. אפשר וצריך להשתמש ב-namespace. אתם יכולים לדלג לפרק הבא. הספר הזה לא מלמד CSS ותת-פרק הבא מלמד CSS בסיסי בלבד, אבל הוא יספק לכם את בסיס הידע הנחוץ כדי שתוכלו המשיך להמשך הלאה.

## CSS בסיסי

CSS מורכב משני חלקים: סלקטור ותכונות. נסתכל למשל על ה-CSS שהבאנו קודם לכן:

```
button {  
  background-color: red;  
}
```

הו הסלקטור, והוא התכונה `background-color: red` הוא הערך. אני מעניק את התכונות לכל האלמנטים שמתאים לסלקטור. במקרה זה אני מעניק את תכונת צבע הרקע האדום לכל ה-buttons בלי יוצא מהכלל.

## סלקטור

סלקטור יכול להיות מכמה סוגים. הוא יכול להיות אלמנטשלם, כמו `button`, `div`, `span` וכו', והוא יכול להיות גם קלאס. קלאס של אלמנט HTML נראה כך:

```
<div className="myDiv">{count}</div>
```

אפשר לתת את הקלאס זהה לנמה וכמה אלמנטים בקומפוננטה. אם אני רוצה לתת את התכונות האלו ל-`div`, אני אעתן אותן באמצעות הסימן נקודה - ". ". כמובן, הסלקטור שלי ייראה כך:

```
.myDiv {
  background-color: red;
}
```

אני יכול לשלב כמה סלקטורים בבת אחת. למשל, אם אני רוצה לתת את התכונה של הרקע האדום גם לכל אלמנט שהקלאס שלו הוא `div` וגם לכל `button`, אני אכתוב אותם עם פסיק - "," מפריד ביניהם.

```
.myDiv, button {
  background-color: red;
}
```

סדר הselקטורים לא משנה בדוגמה זו, אך הוא משמעותי כאשר יש הוראות סותרות של סלקטורים דומים והדף צריך לקבוע קידימות.

אפשר לשלב סלקטורים כדי להגיע לסלקטור מדויק יותר. למשל, אני מעניק את אותו הקלאס גם לחלק מהכפتورים וגם ל-div:

```
<div>
  <button onClick={increaseHandler}>Increase</button>
  <button className="myDiv"
onClick={decreaseHandler}>Decrease</button>
  <button className="myDiv"
onClick={restartHandler}>Restart</button>
  <div className="myDiv">{count}</div>
</div>
```

כך אני יכול לבחור רק את ה-buttons שיש להם קלאס myDiv בתחילת הבא:

```
button.mydiv {
  background-color: red;
}
```

זה בעצם שילוב של שני סלקטורים - גם button וגם div.mydiv. אם אין רוח ביניהם זה אומר גם וגם button וגם div.mydiv.

סלקטור חשוב נוספת הוא סלקטור מבוסס id, אך בעולם של הקומפוננטות וריאקט מומלץ בחום לא להשתמש בו. רק לידע כללי, הסלקטור של id באה עם #. קלומר אם יש לי אלמנט שיש לו id מסוים, למשל:

```
<div id="myDivId">{count}</div>
```

از הסלקטור יהיה:

```
#myDivId {
  background-color: red;
}
```

מן ש-id אמרור להופיע פעם אחת בדף, באתר מבוסס קומפוננטות זה לא רעיון טוב בכלל.

יש סלקטורים נוספים כמו סלקטורים לפי תכונות או לפי תוכן – השמיים הם הגבול, אבל כאמור ניאלץ להסתפק בסלקטורים הבסיסיים. לא נדון בהיררכיה של סלקטורים, שהיא חלק ממשמעותי מאוד מ-CSS.

## תכונות

הסלקטורים קבועים על מי נפעיל את התכונות. התכונות קבועות את העיצוב של האלמנטים: צבע, מידים וafilו התנהגות. התכונות נמצאות בתוך הסוגרים המסורסים שלאחר הסלקטור. הן מורכבות מהתכונה ומהערך שלה. למשל, שם התכונה הוא צבע רקע – background-color והוא red. לכל תכונה יש ערכים שאפשר לחת לה. למשל, התכונה width מקבל ערכים בפיקסלים או ביחידות אחרות.

```
button {
    width: 150px;
}
```

אפשר לתת כמה תכונות שרצוים, ללא הגבלה, כל עוד מקפידים שייהי סימן נקודת-פסיק () לאחר כל תכונה.

```
button {
    background-color: red;
    height: 50px;
    width: 150px;
}
```

הינה רשימה קצרה של תכונות והערכיהם שלhn. בעולם האמיתי יש כמובן הרבה יותר תכונות, אבל כאן נctrיך להסתפק ברשימה קצרה:

הערכיהם שלhn	שם התכונה	תיאור התכונה
מספר + ak, למשל: 100px	height	גובה
מספר + ak, למשל: 100px	width	רוחב
שם הцבע, למשל red, או ערך הצבע בהקס' למשל: #fffffff	color	צבע טקסט
שם הцבע, למשל red, או ערך הצבע בהקס' למשל: #fffffff	background-color	צבע רקע
מספר + ak ארבע פעמים. הערך הראשון קובע את השוליים העליוניים, השני את השוליים מימין, השלישי את התחתוניים והרביעי את השוליים משמאלי. למשל: .10px 10px 10px 10px	margin	שוליים
right או left	text-align	כיוון טקסט בתחום האלמנט

עכשו, כשיש לכם את היסודות לבניית CSS בסיסי, תוכלו ליצור את קובצי ה-CSS שלכם ולבצע להם import או לחלופין לעצב את הקומפוננטה בדרךים אחרות. ניתן גם להשתמש ב-ChatGPT ובכלי בינה מלאכותית נוספים כדי לבנות את העיצוב. יש כמה וכמה דרכים נוספות לעיצוב קומפוננטה בריאקט, אבל כל הדרכים האלה נשענות על ספריות צד שלישי. לריאקט עצמה אין דעה בנוגע לדרכי העיצוב, ובודוקומנטציה הרשמית שלה מאפשרת למצוא את הדרך שפורטה פה. יש דרך נוספת לעיצוב קומפוננטות בשם CSS inline, אך היא אינה מומלצת על ידי הדוקומנטציה ופוגעת בביצועיהם.

פרק 16

# הלאס ריאקטוי



# קלאס ריאקטיבי

עד כה, כל הקומפוננטות שעבדנו עלייהן היו פונקציות. הקומפוננטות האלו נקראות באנגלית functional component והן משתמשות בהוקים על מנת לנצל את הסטייט וגם לדעת מתי הקומפוננטה רונדרה. אבל יש גם קומפוננטות אחרות, קומפוננטות מבוססות קלאס. יש כרגע הנחיה חד משמעית בריект שלא לعباد עם קלאסים והאפשרות הזו תאפשר להתמך בעתיד. כדאי לקרוא את הפרק הזה רק אם אתם עובדים בחברה שבה משתמשים בכך יישן שיש בו קלאס. במידה ולא, אנא עירבו לפרק הבא.

אם הספר זהה היה נכתב בתקופה שריект הייתה בגרסת 15, סביר להניח שהקומפוננטות האלו היו תופסות בו נפח גדול יותר, אבל החל מגרסה 16.8 השימוש בקומפוננטות מבוססות פונקציות הפך לפחות יותר. כרגע אפשר לראות מעבר של יותר ויותר מפתחים ושל ריאקט עצמה לכיוון הקומפוננטות מבוססות הפונקציה. אבל חשוב לדעת שיש קומפוננטות מבוססות קלאס בריект ولو רק בגלל היכולת שלכם לעבוד עם קוד ישן יותר (Legacy Code).

אז איך זה נראה? פשוט מאד – עם קלאס. קלאס (Class) הוא חלק מג'אוועסקריפט עצמה ומדובר בסינטקס שונה מפונקציה. לקלאס יש כמה חלקיים – ראשית, יש לו קונסטרקטור (constructor), הפונקציה הבנאית, שפועלת מיד בהתחלת ושאליה אני גם מקבל את ה-props. הקוד בקונסטרקטור מתבצע ברגעור הראשוני בלבד. שנית, יש לו render, שיש בו מה שהקומפוננטה מחזירה.

על מנת שהקלאס יהיה קומפוננטה ריאקטיבית, אנו עושים לו extend לקלאס של .React.Component

הבה נדגים עם קומפוננטה ממש פשוטה, בرمת ה-Hello World. אם הייתי מבקש מכם ליצור קומפוננטה כזו עם פונקציה, סביר להניח שהייתם יוצרים את הקומפוננטה הבאה:

```
function Welcome(): JSX.Element {
  return (
    <div>Hello World!</div>
  );
}

export default Welcome;
```

המקבילה שלה בקלאס תיראה כך:

```
class Welcome extends React.Component {
  render() {
    return <div>Hello World!</div>;
  }
}

export default Welcome;
```

אפשר לראות כאן שmethodת `render`, שיש לנו אותה כיון שהקלאס שלנו יירוש מה-`React.Component`, אחראית על החזרת ה-JSX. עד כאן אין הבדל משמעותי בין קומפוננטת קלאס לקומפוננטת פונקציה, שאוותה אלו מכיריים.

העניין נותרים פשוטים גם אם אנו מתייחסים להשתמש ב-`props`. ה-`props` בקלאס נמצאים על גבי הסkop של הקומפוננטה, כמו למשל `this.props`. בכל מקום נתון בקומפוננטת הקלאס אני יכול להשתמש ב-`this.props` ולקבל גישה ל-`props`. כדי להדגים, הבה נניח שאנו רוצים להוסיף את שם המשתמש ולקבל אותו מה-`props`. בקומפוננטת פונקציה נעשה שהוא זהה:

```

type propsInfo = {
  name: string;
}

function Welcome(props:propsInfo):JSX.Element {
  return (
    <div>Hello {props.name}!</div>
  );
}

export default Welcome;

```

בקלאס אנו ניגשים אל ה-`props` באמצעות ה-`this`. אבל למעשה העובדה הזאת, אין שינוי מהותי. על מנת למשם את אותה התנהגות בדיק עם קלאס, אני אכתוב את הקוד הבא:

```

class Welcome extends React.Component {

  render() {
    return <div>Hello {this.props.name}!</div>;
  }
}

export default Welcome;

```

בקלאס יש לי קונסטרוקטור, שבו אני יכול להשתמש על מנת לעשות פעולות שונות ברגע של הקומפוננטה, לקרוא ל-API חיצוני או להציג מידע בסטייט. בקומפוננטות מבוססות קלאס מקובל מאוד לא להציג מידע ישירות על ה-`this` אלא להשתמש בסטייט.

הבה נניח שאנו רוצים ליצור סטיטו עם ברכה – קלומר לקבל את ה-`props.name`, ליצור ברכה ולהכניס אותה לסטיטו. אם אני ארצה לעשות את זה עם פונקציה, רק לשם הדוגמה, אני אעשה משהו כמו:

```
import { useState } from 'react';

type propsInfo = {
  name: string;
}

function Welcome(props:propsInfo):JSX.Element {
  const [greeting, setGreeting] = useState<string>(`Hello ${props.name} !`);

  return (
    <div>{greeting}</div>
  );
}

export default Welcome;
```

איך נשתמש בסטייט בקלאס? במקרה זהה נדרש להשתמש בקונסטרוקטור, שבו נאתחל את הסטייט ונכנסים לתוכו ערך ראשוני. הסטייט הוא אובייקט שיושב בסkop של הקומפוננטה (כלומר תחת `this`). אין לנו הוק במקרה זהה וכשאנו רוצים להשתמש בסטייט בקלאס, אנו פשוט מגדירים אותו בקונסטרוקטור. קומפוננטת הקלאס שלנו תיראה כך:

```
class Welcome extends React.Component <{name: string}, {greeting: string}>
{
  constructor(props: {name: string}) {
    super(props);
    this.state = { greeting: `Hello ${props.name}!` }
  }

  render() {
    return <div>{this.state.greeting}</div>;
  }
}

export default Welcome;
```

אפשר לראות שהשתמשתי בקונסטרוקטור. הקונסטרוקטור מקבל `props` וגם מפעיל את `super`. השורה הזו חשובה ואפילו קריטית כיון שהיא קוראת לكونסטרוקטור של `React.Component`. זו לא המקרה של ריאקט אלא חלק מהסינטקס של ג'אווהסקריפט. בלי השורה הזו, הקונסטרוקטור שלכם לא יעבוד כשרה.

השורה השנייה היא יצירת הסטייט. אני פשוט יוצר סטייט על `this` ומשתמש בו. אנו בקונטקסט של קלאס, וזה הסיבה לכך שאנחנו משתמשים ב-`this`.

אין לנו הוקים ואני לא מגדירים פונקציית שינוי סטיט. הפונקציה שאנו משתמשים בה לשינוי הסטיט הוא `setState` שמקבלת אובייקט סטיט. השימוש שלה יוארה כך:

```
this.setState({ greeting: `Hello ${props.name}!` });
```

זה נראה מורכב יותר מאשר המימוש הנוכחי והנקי יותר של הפונקציה, זה נcone. מכיוון שיש לנו לפחות, אנחנו נכנסים גם לביעיות של קונטקסט.

הבה נדגים על ידי המרת הקומפוננטה הזו, שהיא פונקציה המשמשת באירועים, לפחות:

```
import { useState } from 'react';
function Counter():JSX.Element {
  const [count, setCount] = useState<number>(0);

  function increaseHandler():void{
    setCount(count + 1);
  }

  function decreaseHandler():void {
    setCount(count - 1);
  }

  function restartHandler():void {
    setCount(0);
  }

  return (
    <div>
      <button onClick={increaseHandler}>Increase</button>
      <button onClick={decreaseHandler}>Decrease</button>
      <button onClick={restartHandler}>Restart</button>
      <div>{count}</div>
    </div>
  );
}

export default Counter;
```

זו קומפוננטת פונקציה עם אירועים שעשינו באחד התרגילים בפרק על קומפוננטות ואירועים. אם נרצה להמיר אותה לפחות, ראשית נזכיר לייזור קונסטרקטור ולשנות את הסטייטים, אבל יש לנו גם אירועים, ולאירועיםanno חיבים להעביר קונטקט של `this`. אחרת, בשעת הפעלת האירוע תעבור הפונקציה בסקוופ אחר (הסקוופ של החלון) ולא תהיה לה גישה לסטיט.

```
interface CounterState {
  count: number;
}
interface CounterPropsInfo {
// props of Counter
}
class Counter extends React.Component<CounterPropsInfo,
CounterState> {
  constructor(props: CounterPropsInfo) {
    super(props);
    this.state = { count: 0 };
    this.increaseHandler = this.increaseHandler.bind(this);
    this.decreaseHandler = this.decreaseHandler.bind(this);
    this.restartHandler = this.restartHandler.bind(this);
  }
  increaseHandler() {
    const newCount = this.state.count + 1;
    this.setState({ count: newCount });
  }
  decreaseHandler() {
    const newCount = this.state.count - 1;
    this.setState({ count: newCount });
  }
  restartHandler() {
    this.setState({ count: 0 });
  }
  render() {
    return <div>
      <button onClick={this.increaseHandler}>Increase</button>
      <button onClick={this.decreaseHandler}>Decrease</button>
      <button onClick={this.restartHandler}>Restart</button>
      <div>{this.state.count}</div>
    </div>
  }
}
export default Counter;
```

צריך לשום לב הוטב לשורות האלו בקונסטרוקטור:

```
this.increaseHandler = this.increaseHandler.bind(this);  
this.decreaseHandler = this.decreaseHandler.bind(this);  
this.restartHandler = this.restartHandler.bind(this);
```

למה צריך אומן? מסיבה פשוטה מאוד – אם האירוע מופעל, למשל האירוע זהה:

```
button onClick={this.restartHandler} >Restart</button>
```

הסקופ הוא לא של הקלאס. על מנת לגרום לאירוע לעבוד בתחום הקלאס עם גישות למשתנים, למתחודות וכמובן לסטיטיט, יש צורך להשתמש ב-bind. ה-bindגורם לאירוע לעבוד בסקופ של הקלאס. אם ה-bind לא יהיה קיים, אני אקבל שגיאה כזו:

TypeError: Cannot read property 'setState' of undefined

כי בשעת הפעלת האירוע, ה-this הוא של החלון ולא של הקלאס.

הבה נסכם את מה שלמדנו עד עכשיו על קלאס ועל פונקציה:

קלאס	פונקציה	פעולה
באמצעות render שבו יש return	באמצעות return	החזרת פלט
באמצעות props שמתקבלים בקונסטרוקטור בקלאס	באמצעות useState שמתקבלים בפונקציה	קבלת קלט
באמצעות: 1. הגדרת state. בקונסטרוקטור. 2. שימוש ב setState	באמצעות הhook useState	ניהול סטייט פנימי
באמצעות מתודות. יש לבצע bind ל-this בקונסטרוקטור לכל אירוע	באמצעות תת-פונקציות בתוך הפונקציה	ניהול אירועים

קל לראות שקלאס הוא מסובך יותר מבחןת כמהות הקוד שיש לכתוב (אם כי יש מי שיגידו שהוא קל יותר להבנה). כרגע המגמה בריאקט היא להשתמש כמה שיותר בקומפוננטות מבוססות פונקציות ופחות בקומפוננטות מבוססות קלאס. לפיכך, הדוגמאות ימשיכו להיות בפונקציות.

## מעגל החיים ב-**קלאס**

ב-**קלאס** ניתן גם לשלוט ב-**מעגל החיים** של הקומפוננטה. בעוד ב-**פונקציה** יש לנו רק את הhook **useEffect**, שפועל כאשר הקומפוננטה מתרנדרת בפעם הראשונה וכאשר היא מתרנדרת מחדש. ב-**קלאס** יש לנו שליטה מלאה יותר ב-**מעגל החיים** וקיימות לא פחות מושגheiten לעשות זאת.

### **componentDidMount**

מתודת המופעלת מייד לאחר שהקומפוננטה נטענת לתוך ה-DOM. זה מקום מעולה לבצע קריאה ל-**API** ופעולות נוספות הקשורות לתחול הקומפוננטה (כמו לוגינג) או להירשם לאירועים, כפי שנראה בדוגמה שבמהלך הפרק.

### **componentDidUpdate (prevProps, prevState, snapshot)**

מתודת המופעלת ברגע שה-**props** מתעדכנים או ה-**setState** מופעל. אנו מקבלים גישה ל-**props** הקודמים ול-**props** הנוכחיים (אחרי העדכון). הפרמטר השלישי הוא נדייר יותר ורלוונטי לשימוש ב-**getSnapshotBeforeUpdate** ולא נדון בו.

**שימוש לב:** בנגד **componentDidMount**, שנקרא רק פעם אחת ב-**מעגל החיים** של הקומפוננטה, **componentDidUpdate** נקרא לאחר כל פעם שמתבצע רנדור (חוץ מהפעם הראשונה, שבה **componentDidMount** מופעל). כדאי גם לשים לב שמאני שהמתודה זו מופעלת כתוצאה מ-**setState**, אם נשים בתוכה עוד **setState** בלי תנאי, עלולה להיות לנו לולאה אינסופית.

### **componentWillUnmount**

מתודת המופעלת ממש לפני שהקומפוננטה נמחקת מה-DOM. אידיאלית לניקיונות של **interval** או לוגינג.

### **shouldComponentUpdate (nextProps, nextState)**

מתודת שבה אנו יכולים לקבוע אם אנו בכלל רוצים שהקומפוננטה תתרנדר מחדש. היא מופעלת כאשר ה-**props** או ה-**state** מתעדכנים. המתודה הזו אמורה להחזיר **true** אם אנו רוצים רנדור חדש או **false** אם אנו לא רוצים רנדור זהה. היא נדרה יחסית ומאפשרת לנו גישה אל ה-**props** שהתעדכנו ועל הסטייט החדש. היא מתרחשת לפני **componentDidUpdate** וכאמור, אם היא מחזירה **false**, אז **componentDidUpdate** לא תרוץ כיון שהקומפוננטה לא תרנדר. התכונון

העתידי של ריאקט הוא לאו דווקא להפוך את זה להוראה אחידה אלא להמלצה למנוע את הרנדור של ריאקט.

היא נדירה יחסית כי ברוב המוחלט של המקרים אנו נשומך על המנווע של ריאקט שיקבע אם קומפוננטה תרונדר או לא. אנו משתמשים בה כאשר נרצה לשפר ביצועים כיון שם חוסכים ברנדור, מקבלים ביצועים משופרים לכל הקומפוננטות.

### **getDerivedStateFromError (error)**

מתודה שמשופעת כאשר אחת הקומפוננטות הבנות זורקת שגיאה ויכולת לעדכן את הסטייט בעקבות השגיאה זו (ולא לבצע פעולות נוספות שיש להן השפעות צדדיות מעבר לשינוי הסטייט). מה שהיא מוחירה מעדכן את הסטייט.

### **componentDidCatch (error, info)**

בדומה ל-`getDerivedStateFromError`, גם המתודה זו מופעלת כאשר אחת מתת-הקומפוננטות זורקת שגיאה, אך היא יכולה גם לקרוא לפונקציות אחרות (כמו `log`). חלקן מיועדות לשימושי קצה, אבל חלקן שימושיות מאוד בקומפוננטות מסויימות. כך, למשל, בקומפוננטות שיש בהן `interval`, מומלץ מאוד להשתמש דווקא בקומפוננטות קלאס כי אפשר לנகוט את ה-`interval` אחר כך ולהקל על הזיכרון. אם לא נעשה זאת, ה-`interval` ימשיך לרווח גם אם הקומפוננטה נמחקה או אם עברנו למקום אחר, וזה יכול לגרום זילגת זיכרון וביעות ביצועים.

כל, למשל, בקומפוננטה `TodayTime`, אני אמחק את ה-`interval` באמצעות פונקציית `מעגל החיים` `:componentWillUnmount`

```
import React from "react";

interface TodayTimePropsInfo {
  interval: Function;
}

interface TodayTimeState {
  time: number;
}

class TodayTime extends React.Component<TodayTimePropsInfo,
TodayTimeState> {
  interval: NodeJS.Timeout | undefined;

  constructor(props: TodayTimePropsInfo) {
    super(props);
    this.state = { time: Date.now() };
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      this.upTime();
    }, 1000);
  }

  componentWillUnmount() {
    if (this.interval) {
      clearInterval(this.interval);
    }
  }

  upTime() {
    this.setState({ time: Date.now() });
  }

  render() {
    return <React.Fragment>{this.state.time}</React.Fragment>;
  }
}
```

```
export default TodayTime;
```

הfonkcija `componentWillUnmount` בעצם מופעלת אוטומטית כאשר הקומפוננטה נמחקת מה-DOM, וזה קורה כתוצאה מפעולות שונות, כמו מעבר דף בראוטינג, שעליינו נרחב בשלב מאוחר יותר.

דוגמה טובה נוספת לשימוש במתודות של מעגל החיים היא למשל בתפיסת שגיאות. מקובל מאוד להשתמש בקומפוננטת אב צו, לדוגמה:

```
import React from "react";

interface ErrorBoundaryPropsInfo {
  children: React.ReactNode;
}

interface ErrorBoundaryState {
  hasError: boolean;
}

class ErrorBoundary extends React.Component<ErrorBoundaryPropsInfo, ErrorBoundaryState> {
  constructor(props:ErrorBoundaryPropsInfo) {
    super(props);
    this.state = { hasError: false };
  }
  componentDidCatch(error:Error, info:React.ErrorInfo) {
    // LOG THE ERROR
  }
  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
export default ErrorBoundary;
```

unin בקומפוננטה מראה שהוא תופסת כל שגיאה ומציגה אותה. אם לא – היא מציגה את מה שיש בתוכה. זה מקובל מאד ואת זה, נכון לגרסה الأخيرة של ריאקט, אפשר לעשות רק בקלאס.

**תרגיל:**

צרו קומפוננטה מבוססת קלאס שבה יש ספרה, החל מ-0, וכפטור. לחיצה על הכפטור מעלה את הספרה ב-1.

**פתרון:**

```
import React from "react";

interface TodayTimePropsInfo {
  interval: Function;
}

interface TodayTimeState {
  time: number;
}

class TodayTime extends React.Component<TodayTimePropsInfo,
TodayTimeState> {
  interval: NodeJS.Timeout | undefined;

  constructor(props: TodayTimePropsInfo) {
    super(props);
    this.state = { time: Date.now() };
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      this.upTime();
    }, 1000);
  }

  componentWillUnmount() {
    if (this.interval) {
      clearInterval(this.interval);
    }
  }

  upTime() {
    this.setState({ time: Date.now() });
  }
}
```

```
render() {
  return <React.Fragment>{this.state.time}</React.Fragment>;
}

export default TodayTime;
```

אין כאן מהهو שאינכם מכירם. את הסטייט אנו מגדירים כבר בקונסטרקטור ואנו מתחלים אותו כ-0, ויצרנו handler לקליק שמעלה את הסטייט ב-1.  
כיוון שהאירוע פועל בסקובף אחר, אנו חייבים לקשרו אליו את הסקופ שלנו באמצעות:

```
this.clickHandler = this.clickHandler.bind(this);
```

אחרת נקבל שגיאות TypeError: Cannot read property 'setState' of undefined בכל פעם שנפעיל את האירוע, כיון שה-this שיש באירוע אינו ה-this של הקלאס.

**תרגיל:**

מנהל המוצר של בזק פנה אליכם בבקשתו ליצור קומפוננטה שמציגה התקפות סייבר. המספר שלתן מתחילה ב-1,000 ועולה ב-10 בכל שנייה. עלייכם למשתמש עם קלאס כיוון שתפקידם נדרש להשתמש ב-setInterval ויש צורך לנוקוט אותו עם clearInterval.

**פתרונות:**

```
import React from "react";

interface CurrentCyberAttackCounterPropsInfo {
    // props of CurrentCyberAttackCounterPropsInfo
}

interface CurrentCyberAttackCounterState {
    attackNumber: number;
}

class CurrentCyberAttackCounter extends React.Component<
    CurrentCyberAttackCounterPropsInfo,
    CurrentCyberAttackCounterState
> {
    interval: NodeJS.Timeout | undefined;

    constructor(props: CurrentCyberAttackCounterPropsInfo) {
        super(props);
        this.state = {
            attackNumber: 1000,
        };
        this.timerTick = this.timerTick.bind(this);
    }
    timerTick() {
        this.setState({ attackNumber: this.state.attackNumber + 10 });
    }
    componentDidMount() {
        this.interval = setInterval(this.timerTick, 1000);
    }
    componentWillUnmount() {
        clearInterval(this.interval); // Cleaning phase
    }
    render() {
```

```
        return <div>{this.state.attackNumber}</div>;
    }
}

export default CurrentCyberAttackCounter;
```

אנו יודעים שאנו צריכים להשתמש בקלאס אם יש צורך להשתמש בניקיון אחרי שהקומפוננטה נעלמת.

ראשית, אנו משתמשים בקלאס כרגע. כיוון שיש לי צורך בזיכרון פנומי, אני יודע שאני צריך סטיט וANI מגדר סטיט כבר בקונסטרוקטור באמצעות `this.state`.

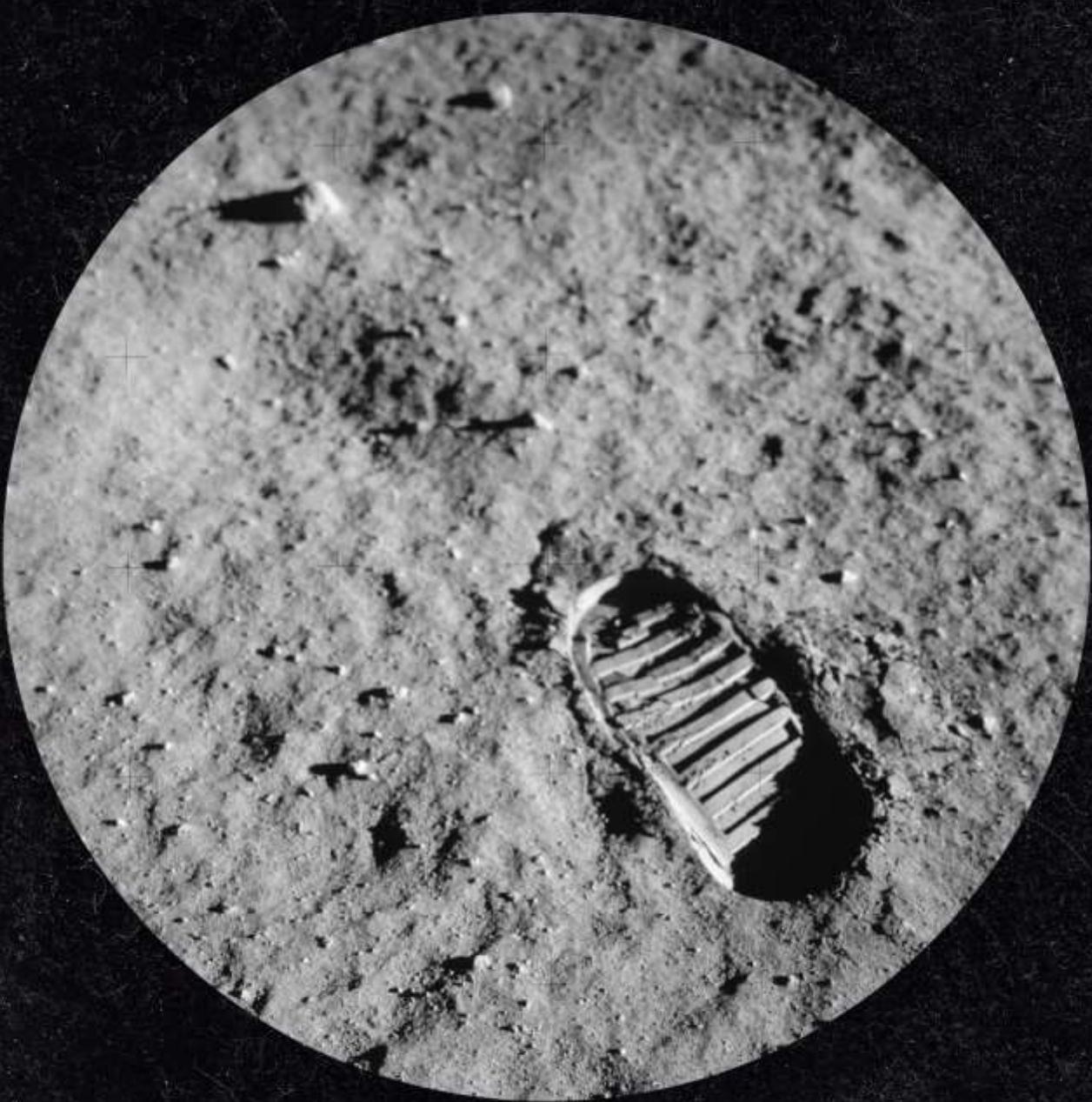
הפעולה השניה היא להגדיר `interval` שיפעל בכל שנייה, ואת זה אני עושה במתודת מעגל החיים `componentDidMount`, שעובדת מייד לאחר שהקומפוננטה מופעלת. אני מייצר `interval` ויוצר לו רפרנס במשתנה שנמצא בסkop של הקומפוננטה. מה שה-`interval` עושה הוא לעדכן את הסטיט `-10`. על מנת שתיהו לו גישה לסטיט, אני מפעיל:

```
this.timerTick = this.timerTick.bind(this);
```

את הניקיון אני עושה בקומפוננטת מעגל החיים `componentWillUnmount`, ואת הפלט אני מגדר ב-`render`.

פרק 17

# שימוש בהומפוניות סמליקות אחרים



## שימוש בקומפוננטות אחרות

אחד מהחווקות הנגדولات של ריאקט היא שימוש בספריות אחרות של קומפוננטות. וכך בעצם אנחנו לא נדרשים לבנות מASF את האפליקציות שלנו אלא פשוט להשתמש בקומפוננטות בקוד פתוח שאחרים בנו וכך אנחנו יכולים להגיע לתוצאות טובות מאוד בזמן קצר יותר. יש בחוץ אינספור קומפוננטות וואוסף קומפוננטות שקל לקחת וליצור איתם אפליקציות מורכבות ביותר ויפות מאוד מבחינה ויזואלית.

הספריות האלה באוט כabilות תוכנה שאנו יכולים להתקין בפרויקט שלנו, שיצרנו ב-Vite. הפרויקט הזה מכיל לא מעט כabilות תוכנה, כמו babel למשל, והcabils המהוות על ידי מנהל התוכנה של Node.js (ג'אויהסקריפט בסביבת השרת) שנראה זקח. לא נדרש לדעת Node.js כדי להשתמש בו-זקח ולהתקין כabilות תוכנה (או להסיר אותן). על מנת להתקין כabilת תוכנה הכוללת קומפוננטות, אנו צריכים לשורת הפקודה שעלייה למדנו בפרק על בניית סביבת עבודה מתקדמת. אנו נפתח את Visual Studio Code, נחפש את לשונית Terminal וナルץ על New Terminal וナルץ על终端. ייפתח לנו מסך של שורת פקודה בתחתית העורך ונוכל לבצע התקנה באמצעות זקח.

לחופין, אם אין לכם, מסיבה מסוימת, Visual Studio Code, אתם יכולים, אם יש לכם מחשב מבוסס חלונות, הגיעו למצב cmd ואז לנOOT באתmoz הפקודה cd אל תיקיית הפרויקט שלכם. גם שם אפשר לבצע התקנה צזו.

כדי להתקין קומפוננטות אחרות אנחנו צריכים לדעת אילו קומפוננטות אלו רוצים. יש אולי אוסף קומפוננטות וקומפוננטות שונות. מקור טוב לחפש הוא גול. כאן נתרgal באתmoz UI React Material – אוסף קומפוננטות בעיצוב חברות גול שיצר המון קומפוננטות בסיסיות בריект. העיצוב הזה נפוץ מאוד ולא מעט אפליקציות וbsites ואתרים משתמשים באוסף הזה. כתובות הפרויקט היא:

<https://material-ui.com/>

מיד בדף הראשון נבחר ב-get started וניתנה שיס לנו הוראות מדויקות להתקנה עם וווקט. נעקוב אחריה. בחולון הטרמינל שלנו נקליד:

```
npm install @mui/material @emotion/react @emotion/styled
```

הפקודה זו משתמשת ב-wook על מנת להוריד ולהוסיף לפרויקט באופן אוטומטי את ספריית UI React Material על הקומפוננטות שלה.

זה כל מה שנדרש! לאחר המתנה קצרה נקבל פלט שנראה כך:

```
PS C:\Users\barzik\local\my-app> npm install @material-ui/core
npm WARN @typescript-eslint/eslint-plugin@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @typescript-eslint/parser@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN ts-pragm@1.2.2 requires a peer of typescript@^2.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN tslint@5.27.1 requires a peer of typescript@>2.8.0 || >= 3.2.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\jest-haste-map\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.1.7 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.1.7: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ @material-ui/core@1.6.1
added 31 packages from 41 contributors and audited 903760 packages in 17.413s
```

עכשו אפשר להשתמש בקומפוננטות הelow.

**חשיבות:** בשנים האחרונות הפכו js.js ו-wook לסטנדרט של ממש בכל הקשרו לסביבות פיתוח וכל הספריות המודרניות משתמשות ב-wook להתקנה ולניהול. ספר זה אינו מלמד וווק אך הוא קל להפעלה. להלן כמה פקודות שימושיות בטרמינל. יש להקליד אותן מהתיקייה שבה הפרויקט נמצאו:

## הפקודה

```
npm install PACKAGE_NAME
```

## תיאור הפעולה

התקנה של חבילת תוכנה

```
npm uninstall PACKAGE_NAME
```

הסרת התקנה של חבילת תוכנה

```
npm ls PACKAGE_NAME
```

בדיקה אם חבילת תוכנה מותקנת אצלכם

או

בדיקה של הקובץ package.json

על מנת להשתמש בקומponeנטות האלו צריך רק להשתמש ב-import, כמו בכל קומponeנטה. ההבדל היחיד הוא שהוא שאנו לא נדרשים לבצע import מורכב עם ייבוא מתיקייה אלא import עם שם החבילה בלבד.

אחת הקומponeנטות היא קומponeנטת כפטור בשם Button, שבעצם יוצרת כפתורים מעוצבים. הנה נשתמש בה עם הקומponeנטה Counter שבנוינו בעבר:

```
import { useState } from 'react';
import './Counter.css';

function Counter():JSX.Element {
  const [count, setCount] = useState<number>(0);

  function increaseHandler():void {
    setCount(count + 1);
  }

  function decreaseHandler():void {
    setCount(count - 1);
  }

  function restartHandler():void {
    setCount(0);
  }

  return (
    <div>
      <button onClick={increaseHandler}>Increase</button>
      <button onClick={decreaseHandler}>Decrease</button>
      <button onClick={restartHandler}>Restart</button>
      <div>{count}</div>
    </div>
  );
}

export default Counter;
```

אם תרצו את הקומponeנטה זו, תוכלו לראות שהכפתורים הם כפתורים רגילים ומכוונים. הנה ניצור כפתורים מעוצבים יותר. איך? יש בספרייה של UI React Material כפתורים רבים יפים הרבה יותר. שם הרכיב של הכפטור הוא Button. אנו מיבאים אותו באמצעות:

```
import Button from '@mui/material/Button';
```

ומשתמשים בו בדיק כmo בכל קומפוננטה אחרת, ממש כך:

```
return (
  <div>
    <Button variant="contained" color="primary"
onClick={increaseHandler}>Increase</Button>
    <Button variant="contained" color="primary"
onClick={decreaseHandler}>Decrease</Button>
    <Button variant="contained" color="primary"
onClick={restartHandler}>Restart</Button>
    <div>{count}</div>
  </div>
);
```

ה-import עם הסימן @ עלול לבלב או להטריד בתחילת, אבל זה פשוט שם החבילה (אפשר להשתמש בתווים מיוחדים בשם חבילה). זה הכל!

בדוגמה אפשר לראות שהעברית props בשם variant ו-color משמשת על מראה הcptors. בטעוד של הקומפוננטה של cptor, וכל קומפוננטה אחרת, יש הסברים על ה-props שאפשר להעביר ועל איך הם משנים את המראה של הקומפוננטה.

אפשר, כמובן, להשתמש בכמה קומפוננטות שרצים באותה קומפוננטה, בדוק כמו בקומפוננטה רגילה!

```
import { useState } from 'react';
import TextField from '@mui/material/TextField';
import Button from '@mui/material/Button';

function InputViewer():JSX.Element {
  const [text, setText] = useState<string>('');
  const [viewText, setViewText] = useState<string>('');
  function changeHandler(e: React.FormEvent<HTMLInputElement>):void
{
  setText(e.target.value);
}

  function clickHandler():void {
    setViewText(text);
  }

  return (
    <div>
      <span>{viewText}</span>
      <TextField onChange={changeHandler} />
      <Button onClick={clickHandler}>Click me</Button>
    </div>
  );
}

export default InputViewer;
```

## ייבוא כמה קומפוננטות

אם אני מיבא כמה קומפוננטות מסוימת ספרייה, מקובל להשתמש בסינטקס בסינטקם מעט שונה ליבוא.  
במילים:

```
import TextField from '@mui/material/TextField';
import Button from '@mui/material/Button';
```

כותבים:

```
import { TextField, Button } from '@mui/material';
```

אנו פשוט משתמשים בהמרה של ג'אוועסקריפט - זו צורת כתיב שונה אותו הדבר.

חשוב לציין שהכח האמתי של ריאקט – השימוש בספריות קומפוננטות כללו, שנותנות לנו כוח אדיר רק באמצעות התקנה פשוטה של ספריות קוד פתוח ושימוש קל. באתר של React UI Material יש דוגמאות והסבירים רבים כיצד להשתמש בקומפוננטות, ובזמן קצר מאוד אפשר לבנות דפים מורכבים מאוד.

הבה נדגים עם קומפוננטה אחרת, שבאה על בסיס קומפוננטות אחרות: Grid. אפשר להציג באתר הרשמי של הקומפוננטה – שם יש הוראות התקנה מדיוקות:

<https://devexpress.github.io/devextreme-reactive/react/grid/>

אבל אנו נדגים גם פה – ראשית יש להתקין את הקומפוננטה באמצעות npm באופן הבא:

```
npm install @devexpress/dx-react-core
npm install @devexpress/dx-react-grid
npm install @devexpress/dx-react-grid-material-ui
npm install @mui/icons-material
```

השלב הבא הוא פשוט... להשתמש בה!

```
import { Grid, Table, TableHeaderRow } from '@devexpress/dx-react-grid-material-ui';

function TableViewer():JSX.Element {
    return (
        <Grid
            rows={[
                { id: 0, name: 'Avraham', city: 'Aram Naharaim' },
                { id: 1, name: 'Itzhak', city: 'Desert' },
                { id: 2, name: 'Yaakov', city: 'Tent' },
                { id: 3, name: 'Esav', city: 'Field' },
                { id: 4, name: 'Moshe', city: 'Cairo' },
            ]}
            columns={[
                { name: 'id', title: 'ID' },
                { name: 'name', title: 'Name' },
                { name: 'city', title: 'City' },
            ]}>
            <Table />
            <TableHeaderRow />
        </Grid>
    );
}

export default TableViewer;
```

אם תנסו את הקוד זהה, תראו שנוצרת לכם טבלה מעניינת. יש לא מעט מודולים חיצוניים שיכולים לחסוך לכם עבודה ומאפשרים לכם לקבל תוצאות מהירות ויפות כמעט מיד. שווה להתחיל להתעניין ולנסות, למשל, את UI React Material. יש הרבה דמואים ודוגמאות קוד מעניינות שדרכם יכולים ליצור תוכר מהם כמעט מושך עבודה.

## מודולים חסרים

לעתים, כשתנסו קומפוננטות מסוימות אחרות, אתם עלולים להיתקל בשגיאה נורית:

```
Module not found: Can't resolve '@mui/icons-material/ChevronLeft' in  
C:\PROJECT
```

או בשגיאה דומה. השגיאות הללו מתקבלות כאשר המודול פשוט לא הותקן על ידי ווקח. במקרה זהה פשוט כדאי לנגן את התוצאה או להתקין את הספרייה. בשגיאה זו הספרייה החסורה היא `@material-ui/icons`

```
npm install @mui/icons-material
```

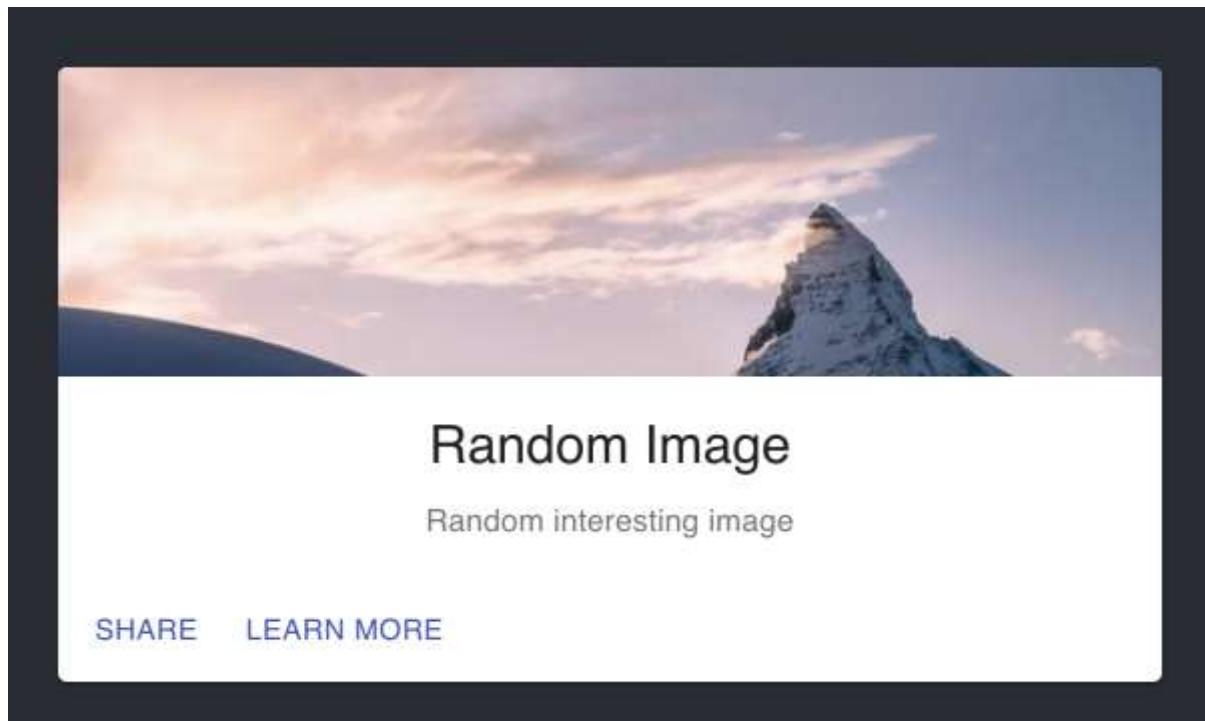
היא מתקינה את הספרייה החסורה שצריך להשתמש בה.

## תרגיל:

באמצעות הרכיב `media card` שיש ב-UI React Material וכן באמצעות הקישור לתמונה רנדומלית:

<https://picsum.photos/id/866/700/400>

צרו את הרכיב זהה:



רמז: התיעוד והדוגמאות לכרטיס נמצאים בכתובות הzon:

<https://material-ui.com/components/cards/>

**פתרונות:**

```

import Card from '@mui/material/Card';
import CardActionArea from '@mui/material/CardActionArea';
import CardActions from '@mui/material/CardActions';
importCardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Button from '@mui/material/Button';
import Typography from '@mui/material/Typography';

export default (): JSX.Element => {
  return (
    <Card sx={{ maxWidth: 345 }}>
      <CardActionArea>
        <CardMedia
          component="img"
          image="https://picsum.photos/id/866/700/400"
          title="Random Image"
          height="140"
        />
        <CardContent>
          <Typography gutterBottom variant="h5" component="h2">
            Random Image
          </Typography>
          <Typography variant="body2" color="textSecondary"
component="p">
            Random interesting image
          </Typography>
        </CardContent>
      </CardActionArea>
      <CardActions>
        <Button size="small" color="primary">
          Share
        </Button>
        <Button size="small" color="primary">
          Learn More
        </Button>
      </CardActions>
    </Card>
  );
}

```

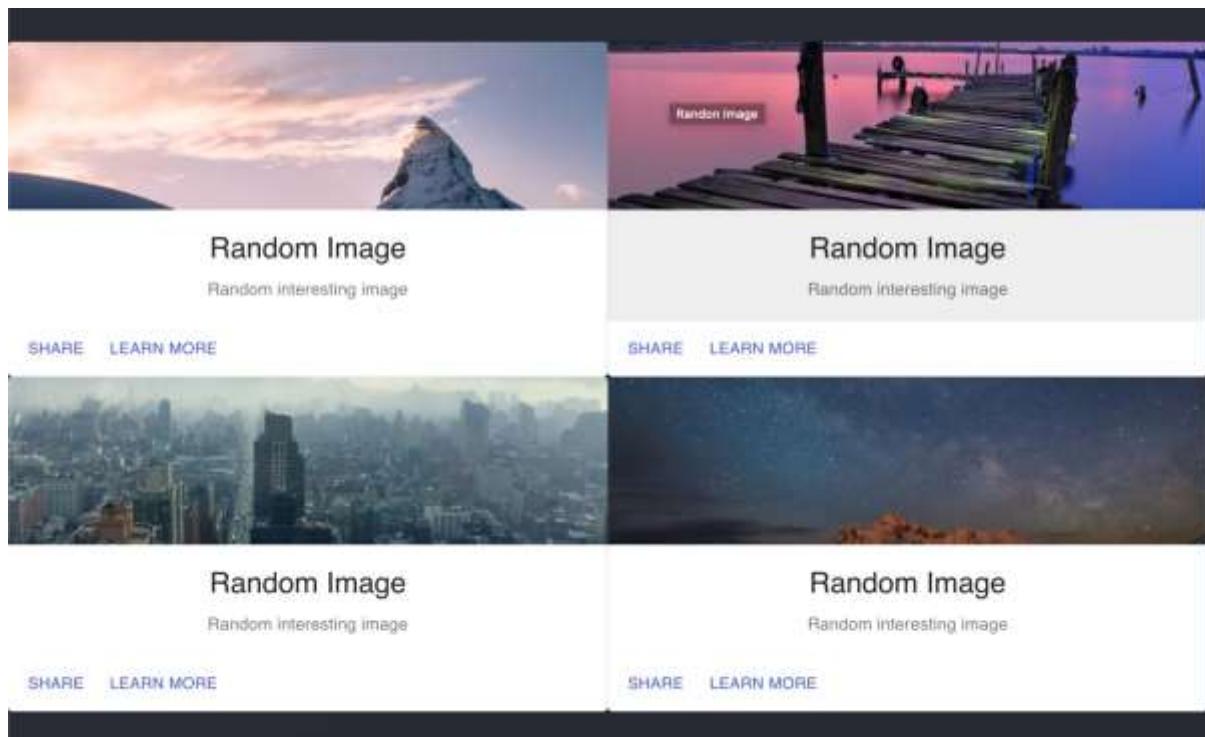
אין כאן גאונות גדולה, פשוט העתקתי את הדוגמה המופיעה בדוקומנטציה כדי ליצור את מה שרציתי ליצור. במקרים האלו אין מה לשבור את הראש ולהתאמץ, אלא צריך פשוט לחת את מה שיש בדוקומנטציה.

כדי לשים לב שהדרך של UI React Material לקביעת עיצוב היא דרך שונה המשמשת `useStyles`, זהה HOC. לא צריך להבין איך זה עובד על מנת להשתמש בהזה, פשוט לזרום עם מה שיש בדוקומנטציה. על HOC נלמד בהרחבה בפרק הבא.

את הקוד הזה אפשר לשימוש ב-`RandomImageCard.tsx`, וזו להשתמש בקומפוננטה זו איפה שרצוים או לדאוג להעביר לה `props` שיקבעו את התמונה השונה.

### **תרגיל:**

באמצעות הרכיב הקודם, צרו ארבעה כרטיסים כאלה:



עם התמונות שבקישורים הבאים:

<https://picsum.photos/id/866/700/400>

<https://picsum.photos/id/867/700/400>

<https://picsum.photos/id/868/700/400>

<https://picsum.photos/id/869/700/400>

**רמז:** העבירו את התמונה לקומפוננטה הקודמת באמצעות props במאזן כפי שלמדו.

**פתרונות:**

אנו צריכים לחת את הקומפוננטה הקודמת, זו שהעתקנו מהדוקומנטציה, ופשוט לשנות אותה כך שתדע לקבל `props`. השינוי זהה הוא פשוט: הקומפוננטה מעבירה `props` והם נכנסים במקום כתובת התמונה בקוד הזה:

## &lt;CardMedia

```
className={classes.media}
image="https://picsum.photos/id/866/700/400"
title="Random Image"
/>>
```

אמנם הקומפוננטה נראהיה מורכבת, אבל השינוי הזה אינו מורכב:

```
import Card from '@mui/material/Card';
import CardActionArea from '@mui/material/CardActionArea';
import CardActions from '@mui/material/CardActions';
importCardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Button from '@mui/material/Button';
import Typography from '@mui/material/Typography';

export default (props :{imageSrc:string}):JSX.Element => {
  return (
    <Card sx={{ maxWidth: 345 }}>
      <CardActionArea>
        <CardMedia
          component="img"
          image={props.imageSrc}
          title="Random Image"
          height="140"
        />
        <CardContent>
          <Typography gutterBottom variant="h5" component="h2">
            Random Image
          </Typography>
          <Typography variant="body2" color="textSecondary"
component="p">
            Random interesting image
          </Typography>
    
```

```

        </CardContent>
    </CardActionArea>
<CardActions>
    <Button size="small" color="primary">
        Share
    </Button>
    <Button size="small" color="primary">
        Learn More
    </Button>
</CardActions>
</Card>
);
}
}

```

השורה שהשתנתה היא:

```
image={props.imageSrc}
```

כעת רק נותר להעביר לקומפוננטה את הפרמטרים האלה ולעשות את זה ארבע פעמים בקומפוננטה מסוימת, למשל ב-**:MyContainer**:

```

import RandomImageCard from './RandomImageCard';
import { Grid } from '@mui/material';

function MyContainer():JSX.Element {
    return (
        <Grid container xs={12}>
            <RandomImageCard
                imageSrc="https://picsum.photos/id/866/700/400" />
            <RandomImageCard
                imageSrc="https://picsum.photos/id/867/700/400" />
            <RandomImageCard
                imageSrc="https://picsum.photos/id/868/700/400" />
            <RandomImageCard
                imageSrc="https://picsum.photos/id/869/700/400" />
        
```

```
</Grid>
);
}

export default MyContainer;
```

השתמשתי כאן ב-Grid, שגם היא קומפוננטה של UI React Material, כדי לסדר את העיצוב. תוכלו לבדוק אותה בדוקומנטציה. אם השתמשת ב-CSS זה מעולה, אבל קל יותר להשתמש בקומפוננטה מסודרת שהיא כבר יצר.

כל שנוטר לי הוא לחת את קומפוננטת MyContainer ולהציג אותה איפה שאני רוצה, למשל ב-`app`:

```
import './App.css';
import MyContainer from './MyContainer';

function App():JSX.Element {
  return (
    <div className="App">
      <header className="App-header">
        <MyContainer />
      </header>
    </div>
  );
}

export default App;
```

ועכשיו אני יכול לבחון בהנאה את התוצאה.  
לא צריך ליצור כל קומפוננטה בלבד. מומלץ מאוד וגם צריך להשתמש בקומפוננטות של אחרים כדי ליצור ממשקים מהמים. לכל קומפוננטה וקומפוננטה יש API משלה. כך, למשל, ב-React UI Material אפשר להעביר `onClick` כדי לבצע פעולה מסוימת, כמו לפתח טאב חדש.

**תרגיל:**

נוסף על פרמטר התמונה, העבירו גם פרמטר של קישור. לחיצה על Learn More תפתח את הקישור הזה.

**רמז:** פתיחת קישור בלשונית דפדן חדשה נעשית באמצעות:

```
window.open(props.linkTo);
```

**פתרונות:**

```
import Card from "@mui/material/Card";
import CardActionArea from "@mui/material/CardActionArea";
import CardActions from "@mui/material/CardActions";
import CardContent from "@mui/material/CardContent";
import CardMedia from "@mui/material/CardMedia";
import Button from "@mui/material/Button";
import Typography from "@mui/material/Typography";

export default (props: { linkTo: string; imageSrc: string }): JSX.Element => {
  function goTo(): void {
    window.open(props.linkTo);
  }

  return (
    <Card sx={{ maxWidth: 345 }}>
      <CardActionArea>
        <CardMedia
          component="img"
          image={props.imageSrc}
          title="Random Image"
          height="140"
        />
        <CardContent>
          <Typography gutterBottom variant="h5" component="h2">
            Random Image
          </Typography>
          <Typography variant="body2" color="textSecondary" component="p">
            Random interesting image
          </Typography>
        </CardContent>
    </Card>
  );
}
```

```

    </CardActionArea>
    <CardActions>
        <Button size="small" color="primary">
            Share
        </Button>
        <Button onClick={goTo} size="small" color="primary">
            Learn More
        </Button>
    </CardActions>
</Card>
);
};

```

אנו צריכים לזכור לא להתבלבל ולא לפחות משפט הקומפוננטות שיש פה. יש כפתור? מצוין, אפשר להעביר אליו onClick, בדיק כפי שלמדנו. הכפתור, שהוא רכיב של UI, React Material, יודע יפה מאוד לטפל באירוע. אנו מעבירים לו אירוע פשוט שפותח מה שאנו מעבירים לו בו-props. במקרה זהה:

```

function goTo():void {
    window.open(props.linkTo);
}

```

כל מה שעשינו לעשות הוא להעביר לקומפוננטה שלנו את הקישור. במקרה הזה הגדרתי שהפרמטר יהיה `linkTo`.

```

<Grid container item xs={12}>
    <RandomImageCard LinkTo="https://internet-israel.com"
imageSrc="https://picsum.photos/id/866/700/400" />
    <RandomImageCard LinkTo="https://hebdevbook.com"
imageSrc="https://picsum.photos/id/867/700/400" />
    <RandomImageCard LinkTo="https://he.wikipedia.org"
imageSrc="https://picsum.photos/id/868/700/400" />
    <RandomImageCard LinkTo="https://haaretz.co.il"
imageSrc="https://picsum.photos/id/869/700/400" />
</Grid>

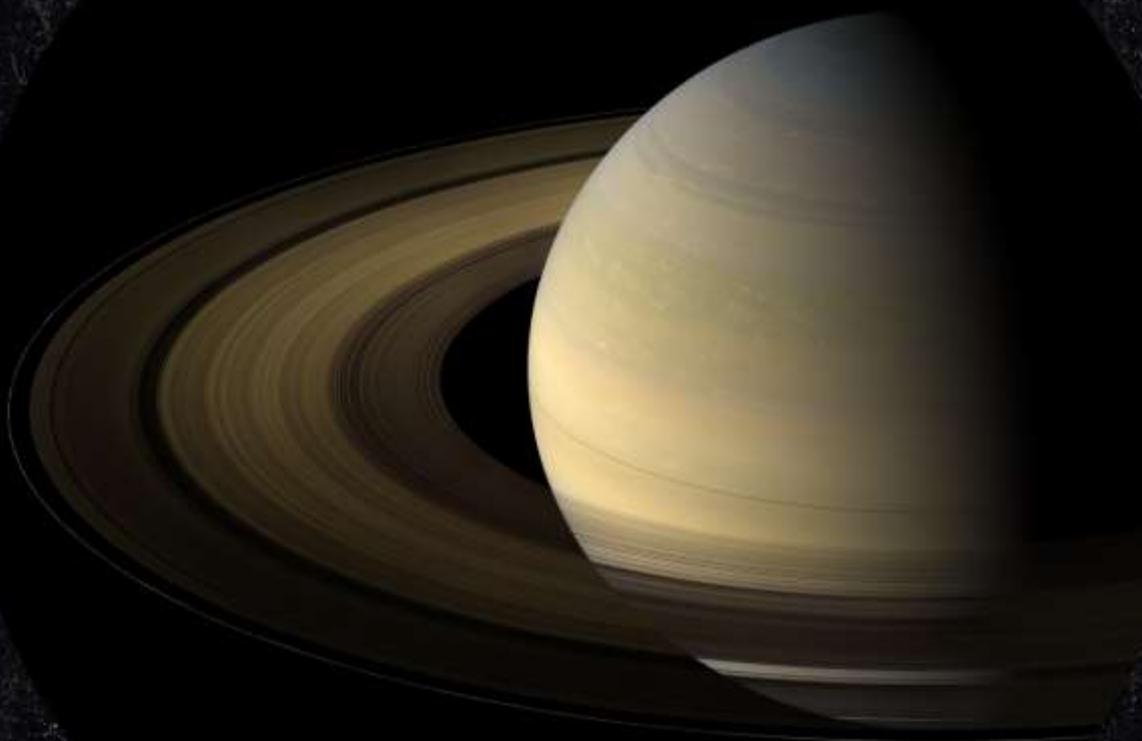
```

כך בעצם אני יכול להמשיך ולפתח על בסיס הקומפוננטה.

שימוש לב שמקובל, במקומן לחזור על קוד, להשתמש בפונקציית `map`.

פרק 18

# HOC: HIGHER ORDER COMPONENT



# HOC: Higher Order Component

מדובר בפיזר של ריאקט שבאמצעותו אנו לוקחים קומפוננטה אחת ומעשירים אותה ביכולות נוספות ללא צורך להכיר את הקומפוננטה המועשת. העשרה זו מתבצעת באמצעות פונקציה עוטפת שמקבלת כารוגומנט קומפוננטה. הפונקציה שמבצעת את השינוי נקראת **HOC**, ראשי תיבות של **Higher Order Component**. המשמעות של המושג זהה היא "קומפוננטות מדרג גבוה" – כלומר ככלו שלא מיליכות את הידים בפונקציונליות אלא מנפקות קומפוננטות אחרות, שהן אלו שעושות את העבודה.

האמת היא שכמתכני ריאקט מתחילה לא יצא לכם לנכתב הרבה HOC, אך בוודאי יצא לכם להשתמש בהם והשימוש בהם עלול להיות מבלבל או סתום יכול להיראות כמו וודו – ככלומר CIS שאל ברור איך הוא עובד. בשנים האחרונות HOC הפכה להיות פחות ופחות פופולרית בגלל הסיבה זו.

כשאנו כותבים HOC, אנו צריכים לזכור שהקלט הוא קומפוננטה והפלט הוא קומפוננטה אחרת שמשתמשה בקומפוננטה של הקלט כבסיס. משהו בסגנון זהה:

```
const HighOrderComponent = (WrappedComponent:JSX.Element) => {
  function HOC():JSX.Element {
    // Stuff
    return <WrappedComponent />;
  }
  return HOC;
};

export default HighOrderComponent;

const SimpleHOC:JSX.Element = HighOrderComponent(MyComponent);
```

מה קורה בפונקציה זו? כיצד אני מעשיר את הקומפוננטה? הבה נדגים עם קומפוננטה שאני רוצה להעшир אף וرك בתוכונה אחת שאני רוצה להעביר לה. איזו? משמעות החיים, הקיום וכל השאר. ערך התוכונה זו, כפי שידוע כל מי שקרא את הספר "מדריך הטרםפיסט לגלקסיה", הוא 42. איך אני מעשיר את הקומפוננטה שלו? אניעביר אליה את כל ה-props שהיא מקבלת (אני לא יודע אילו) וגם את התשובה לחיום עצמו. בדיקך:

```
const HighOrderComponent = (WrappedComponent:JSX.Element) => {
  function HOC(props):JSX.Element {
    const answerToLifeMeaning:number = 42;
    return <WrappedComponent
      {...props}
      theAnswer={answerToLifeMeaning}
    />;
  }
  return HOC;
};

export default HighOrderComponent;

const SimpleHOC: JSX.Element = HighOrderComponent(MyComponent);
```

הדבר שהכי מפחיד מתכנתים צעירים הוא ה-spread operator

{...props}

אבל מה שהוא עושה זה לחת את כל ה-props שמעברים ולהעביר אותם, אין שם, בפורמט מסודר, אל הקומפוננטה שאני עוטף, בלי שאני אctrיך להכניס אותם ידנית. נוסף על כן, אני מעביר גם את משמעות החיים.

זו דוגמה מאוד תיאורטיבית, אבל היא מסיימת להבין עד כמה HOC הם פשוטים למרות הסינטקס המאיים. פשוט לוקחים קומפוננטה אחת ומוסיפים לה תוכנות או מתודות. הבה נראה דוגמה מעשית יותר. באחד הפרקים הקודמים יצרנו קומפוננטה שכילה סטייט עם מספר, שאתחול-

0, ולחיצה על הכפתור גרמה למונה לעלות-1. מה הבעה? בכל פעם שביצענו רענון לדף, הטעית התאפס והמונה חזר ל-1. הטעית נשאר בזיכרון אבל רק לאחר מכן הקומפוננטה. עם HOC אנו יכולים להעшир כל קומפוננטה שהיא ולחסוף API שיבצע שמירה של המספר זהה. הקומפוננטה נראה כך:

```
import { useState } from 'react';
import Button from '@mui/material/Button';

function CounterUp(): JSX.Element {
  const [count, setCount] = useState<number>(0);

  function increaseHandler(): void {
    setCount(count + 1);
  }

  return (
    <div>
      <Button variant="contained" color="primary"
        onClick={increaseHandler}>Increase</Button>
      <div>{count}</div>
    </div>
  );
}

export default CounterUp;
```

אתם יכולים ומוגנים להציב אותה אצלכם ולראות שהיא עובדת באופן מושלם. אך הבעה היא שברגע שאתם טענים מחדש את הדף (באמצעות F5 למשל), המונה מתאפס וחוזר ל-0. אנו רוצים להעшир את הקומפוננטה זו באמצעות HOC על מנת לשמר את המונה בזיכרון הדף דן.

שמירה ב"זיכרון" של הדף, באופן ישיר או גם רענון, נעשית באמצעות שמירה לאובייקט שנקרא `localStorage`. מדובר בקובץ קטן שנשמר על המחשב ושמור נתונים גם לאחר רענון של הדף. הכתיבה אליו מתבצעת באמצעות:

```
localStorage.setItem(key, value);
```

והקריאה מתבצעת באמצעות:

```
localStorage.getItem(key);
```

ראוי לציין שאין שום קשר בין localStorage לריект. מדובר בפיזר של הדף ושל הג'אווסקורייפט שרצה עליו, בדיק כמוך fetch, למשל, המשמש ל-AJAX בסביבת שרת.

פונקציית ה-HOC שלי תינח כל קומפוננטה ותוסיפה לה Methodות save ו-load. לפונקציה אני אקרא :WithStorage

```
const WithStorage = (WrappedComponent:JSX.Element) => {
  function HOC(props):JSX.Element {
    function save(key:string, data:any) {
      localStorage.setItem(key, data);
    }
    function load(key:string):any {
      return localStorage.getItem(key);
    }
    return <WrappedComponent
      {...props}
      save={save}
      Load={load}
    />;
  }
  return HOC;
};

export default WithStorage;
```

מה היא עשויה? מוסיפה לכל קומפוננטה שעוברת דרך שתי מתודות פשוטות: `save` ו-`load`.  
 מתודה `save` שומרת ל-`localStorage` ומתודה `load` טוענת מה-`localStorage`.  
 איך אני מעשיי קומפוננטה זו? אני פשוט זכר שהקומפוננטה שעוברת דרך ה-HOC מקבלת ל-`props` שלא כינה תכונות חדשות, במקרה שלנו `save` ו-`load`.

ראשית, הטעינה עצמה נראה כך:

```
import WithStorage from './WithStorage';
import CounterUp from './CounterUp';

const ComposedComponent:JSX.Element = WithStorage(CounterUp);

function MyContainer():JSX.Element {
  return (
    <ComposedComponent />
  );
}

export default MyContainer;
```

בתוך קומפוננטת אב אני מבצע `import` לשתי הקומפוננטות שלי: ה-HOC והקומפוננטה שעוברת דרךו, ואז אני יוצר משתנה שאליו אני מחזיר את הקומפוננטה המועשרת:

```
const ComposedComponent :JSX.Element = WithStorage(CounterUp);
```

בקומפוננטה זו אני יכול להשתמש כמו בכל קומפוננטה אחרת. צריך לזכור שה-HOC, במקרה זה `WithStorage`, תמיד מחזיר לי קומפוננטה שאני יכול להשתמש בה ב-JSX. זה בדוק מה שאני עושה בקומפוננטת הקונטינר:

```
function MyContainer() :JSX.Element {
  return (
    <ComposedComponent />
  );
}
```

}

از אחרי שהעשתה, אני יכול להשתמש ב-save וגם ב-load בקומפוננטה המקורית שלו. איך פשוט מאד – ה-HOC מוסיף אותו דרך ה-props, אז אני יכול להשתמש בהן דרך ה-props. הנה למשל ה-save:

```
import { useState, useEffect } from 'react';
import Button from '@mui/material/Button';

function CounterUp(props :{save :(key:string, value:number)=>void}) :JSX.Element {
  const [count, setCount] = useState<number>(0);

  function increaseHandler():void {
    setCount(count + 1);
    if (props.save) {
      props.save('counter', count + 1);
    }
  }

  return (
    <div>
      <Button variant="contained" color="primary"
onClick={increaseHandler}>Increase</Button>
      <div>{count}</div>
    </div>
  );
}

export default CounterUp;
```

בכל פעם שה-state מתעדכן, ב-increaseHandler שמו פועל לאחר לחיצה על הכפתור, אני קורא ל-save ומעביר דרכו שני פרמטרים: ה-key שהחלמתי שהוא ה-counter והערך, שהוא מספר.

**שימוש לב:** על מנת למנוע שגיאה אם הקומפוננטה לא עטופה ב-HOC, הוסף תנאי שבודק אם יש לנו `props.save`.

טוב, השמירה היא קלה, אבל מה עם החילוץ? פה אני חייב להשתמש ב-`useEffect`, ההוק שרצ בכל רנדור. אני רוצה שהוא יירוץ אך ורק בrndor הראשון, אז הארגומנט השני הוא מערך ריק. ב-`useEffect`, שקורה רק כאשר הקומפוננטה מתրנדרת לראשונה, אני אקרא למתודת `load` עם המפתח (שהוא `counter`), אמיר אותו במספר (כל הערכיהם ב-`localStorage` נשמרים כמחרוזת טקסט) ואכנסו אותו לסתיט. אם אין כלום ב-`localStorage`, אני מחזיר מספר.

```
import { useState, useEffect } from 'react';
import Button from '@mui/material/Button';

type propsInfo ={
  load: (value: string) => void;
  save: (key: string, value: number) => void;
}

function CounterUp(props: propsInfo) :JSX.Element {
  useEffect(() => {
    let counter:any;
    if(props.load) {
      counter = props.load('counter');
    }
    const initialCounter:number = Number(counter) || 0;
    setCount(initialCounter);
  }, []);

  const [count, setCount] = useState<number>(0);

  function increaseHandler():void {
    setCount(count + 1);
    props.save('counter', count + 1);
  }

  return (
    <div>
      <Button variant="contained" color="primary"
      onClick={increaseHandler}>Increase</Button>
      <div>{count}</div>
    </div>
  );
}
```

```
{
  export default CounterUp;
```

קטע הקוד של `useEffect` עלול לבבל. הבה ננתח אותו שוב. מדובר בפונקציה שמקבלת שני פרמטרים: פונקציה אונומית וערך ריק. המערך הריק נועד להורות לריאקט להפעיל את הפונקציה האונומית רק פעם אחת, בתחילת טיענת הקומפוננטה.

בפונקציה האונומית יש קרייה ל-`props.load`, המתודה השנייה לנו על ידי ה-HOC. אם היא נמצאת אני ממיר אותה למספר. אם לא, אני מחזיר 0 והכל נכנס לסתיט. זה הכל. אני בודק, כמובן, אם `props.load` קיימת.

از למה להיכנס לכל הסרט הזה של ה-HOC ולא פשוט להכניס את ה-`localStorage.set` וה-`localStorage.get` לקומפוננטת `CounterUp` מההתחלה? למה לטrhoה לעטוף אותה ב-HOC? התשובה היא שעכשיו אני יכול לחת את ה-HOC הזה ועתופו אליו קומפוננטות נוספות ולתת להן, כבמטה קسم, את היכולת לקרוא ולכתוב לתוך ה-`localStorage` בלי להתאים כלל ובלוי לנכון שוב ושוב `localStorage.get` ו-`localStorage.set`. וזה דבר חשוב מאוד, כי לעיתים יש לנו פונקציונליות שאנו צריכים להטמע בקומפוננטה שלנו בנוגע לתקשורת לצד השרת זה וחושך המונח-הomon קוד מיותר. הקראות לצד השרת, או במקרה הזה ל-`localStorage`, תמיד יהיו מרכזיות במקום אחד.

**תרגיל:**

הוסיף ל-HOC שכתבנו clear – מתודת clear שמנקה את ה-localStorage – WithStorage – מתחדש localStorage נעשה באמצעות:  
**רמז:** ניקוי ה-localStorage נעשה באמצעות:

```
localStorage.clear();
```

**פתרונות:**

```
const WithStorage = (WrappedComponent: JSX.Element) => {
  function HOC(props: any): JSX.Element {
    function save(key: string, data: any): void {
      localStorage.setItem(key, data);
    }
    function load(key: string): any {
      return localStorage.getItem(key);
    }
    function clear(): void {
      localStorage.clear();
    }
    return (
      <WrappedComponent {...props} save={save} Load={load} clear={clear} />
    );
  }
  return HOC;
};
export default WithStorage;
```

אין בעיה להוסיף כמה פונקציות שנרצה ל-HOC ולהעיר בכך props שבא לנו את clear הקומפוננטות שעוברות דרך ה-HOC. במקרה זה פשוט הוספנו מתודה נוספת nosافت שנקראת clear ועושה את הפעולה הבאה:

```
localStorage.clear();
```

**תרגיל:**

בקומפוננט CounterUp שהודגמה בפרק ועטופה ב-HOC בשם WithStorage, הוסיףו כפתרו שמבצע איפוס של המונה.

**רמז:** איפוס המונה מתבצע באמצעות הפעלת מתודת clear שהוספנו אל WithStorage HOC. יש ליצור כפתרו שיפעל את המתודה זו.

**פתרונות:**

```
import { useState, useEffect } from 'react';
import Button from '@mui/material/Button';

type propsInfo = {
  load:(value:string)=>void,
  save:(key: string, value:number)=>void,
  clear: ()=> void
}

function CounterUp(props:propsInfo):JSX.Element {
  useEffect(() => {
    const initialCounter:number = Number(props.load('counter'))||0;
    setCount(initialCounter);
  }, []);

  const [count, setCount] = useState<number>(0);

  function increaseHandler():void {
    setCount(count + 1);
    if (props.save) {
      props.save('counter', count + 1);
    }
  }

  function resetHandler():void {
    setCount(0);
  }
}
```

```

if (props.clear) {
  props.clear();
}

return (
  <div>
    <Button variant="contained" color="primary"
onClick={increaseHandler}>Increase</Button>
    <Button variant="contained" color="primary"
onClick={resetHandler}>Reset</Button>
    <div>{count}</div>
  </div>
);
}

export default CounterUp;

```

ראשית יש להוסיף כפטור. הכפטור הוא כפטור ה-`:reset`:

```

<Button variant="contained" color="primary"
onClick={resetHandler}>Reset</Button>

```

ב-`onClick` שלו מופעלת הפונקציה `resetHandler`. הוא תפעיל בתורה את ה-`props.clear` וגם תאפס את הסטייט:

```

function resetHandler():void {
  setCount(0);
  if (props.clear) {
    props.clear();
  }
}

```

מאיופה ה-props.clear מגיע? מה-HOC שעוטף את הקומפוננטה. זו הסיבה שאנו בודקים אם הוא קיים. איפה הוא עוטף את הקומפוננטה הזו? הראיינו בפרק. בקומפוננטה Container:

```
import WithStorage from './WithStorage';
import CounterUp from './CounterUp';

const ComposedComponent: number = WithStorage(CounterUp);

function MyContainer(): JSX.Element {
  return (
    <ComposedComponent />
  );
}

export default MyContainer;
```

פרק 19

# לאוֹטִינְגֶּר



## ראוטינג

עד כה עבדנו עם אפליקציות בננות דף אחד שבו יש קומפוננטות שונות. אבל זה כמובן לא ריאלי. באתר אמיתי, או באפליקציה אמיתי, יש כמה וכמה דפים. באתר אינטרנט בסגנון הישן, כל מעבר דף גרם לטעינה מחודשת של האתר כולו. ריאקט (וכן בכלל אפליקציות הווב המודרניות) מעבורי דפים נועשים ללא טעינה כלל מהשרת ובאמצעות קומפוננטות ריאוטינג שմבוצעות ניתוב של קומפוננטות דרך צד הקלוח, דבר המאפשר חוויה גלית טובה בהרבה. הטכנית הזו נקראת SPA – ראשי תיבות של Single Page Application.

ראוטינג (Routing), או בעברית תקנית "ניתוב", הוא מונח ידוע בכלל הנוגע לאפליקציות וביבות, ומובן שיש אותו גם בריקט. ריאקט, ברגע לפירימורקים אחרים (כמו אングולר), לאקובעת איך למש את הריאוטינג, אלא יש כמה קומפוננטות ניוט חיצונית שמומלצות על ידי ריאקט וכל אחד יכול לבחור מה שהוא רוצה. אפשר גם למש את הריאוטינג בעצמכם, אם כי זה לא מומלץ בהתחשב בעובדה שיש קומפוננטות רבות אחרות שכבר נבדקו במאות ואלפי אתרים אחרים. האובייקט `bagoohashcript` המכיל את הכתובת נקרא `location`. אפשר למצוא אותו תחת `window` וANO יכולם לשנות אותו באמצעות `bagoohashcript`.

הבעיה היא ששינוי שלו גורם לטעינה חדשה זהה, כפי שכבר למדנו, גורם לרנדור חדש של כל הקומפוננטות ויוצר חוות משמש לא טובה, וגם עלול לגרום בעית ביצועים. הפטרון? לשנות את `theLocation` בלי לטען את הדף מחדש. יש שתי דרכים לעשות את זה: באמצעות שימוש באובייקט `History`, שנמצא תחת `window`, או באמצעות תוספת לכתובת בשם `hash`. השימוש ב-`window.history` נחשב למתקדם יותר והשימוש ב-`hash` נחשב פשוט יותר. הנה נדגים ריאוטינג באמצעות `hash` בלבד.

**חשוב:** זה הסימן `#` שנמצא ב-URL. למשל: `example.com#page1`

השרת אינו מסוגל "לראות" את מה שיש מאחורי הסימן `#` וכשאנו משלבים אותו בדף, הוא לא נשלח כלל לשרת. זו הסיבה ששינוי שלו, אפילו ידני, לא יטعن מחדש את הדף. בגלל זה פופולרי מאוד לראוטינג ב-`theLocation` `Single Page Application` כמו באפליקציות ריאקט.

על מנת לתרגל ריאקט נוצר שלוש קומפוננטות פשוטות שנעבור ביניהן. `About`, `Home`, `Help`. בדוגמאות כאן אני מכניס בהן רק כותרת פשוטה, אבל הן יכולות להכיל דפים שלמים וקומפוננטות אחרות, כמו כן.

#### קומponentת `Home.tsx`:

```
function Home(): JSX.Element {
  return (
    <div><h1>Home</h1></div>
  );
}

export default Home;
```

#### קומponentת `About.tsx`:

```
function About(): JSX.Element {
  return (
    <div><h1>About</h1></div>
  );
}

export default About;
```

קומponent **:Help.tsx**

```
function Help():JSX.Element {
  return (
    <div><h1>Help</h1></div>
  );
}

export default Help;
```

כאמור, אלו קומפוננטות פשוטות לשם לימוד ריאקטינג והבנתו בלבד. כדי לבצע ריאקטינג אנו יוצרים קומponent אב שמכילה את סרגל הניווט ואת הקומponentה שת�ען בכל פעם את קומponentה העמוד הפעיל, ככלمر את `About`, `Home`, או את `Help`.

קומponentה האב בעצם נתלית באירוע בשם `hashchange` שנמצא על `window`. האירוע הזה מתרחש בכל פעם שמשהו משנה את הכתובת שיש אחרי #-#. כך, למשל, אם אני נמצא בדף `example.com#page1` – האירוע יופעל כתוצאה מלחיצה על קישור סמוביל אל `#page2`. על מנת לדעת מה הכתובת שאחרי #-# אני משתמש בפקודה:

`window.location.hash.substr(1)`

הפקודה זו מורכבת משני חלקים – החלק הראשון:

`window.location.hash`

מחזיר לנו את כל מה שיש ב-#. במקרה זה `#page2`.

`substr(1)`

החלק השני מוריד את #-# ומותירה אותנו עם `page2`.

**نبין את זרימת המידע בקומפוננטה לפני הכתיבה שלה:**

1. בתחילת הקומפוננטה מחברת לאירוע `hashchange` פעולה. הפעולה מופעלת בכל פעם שנלחץ קישור ל-#.
2. כשהפעולה מופעלת, אנו בוחנים את `window.location.hash` כדי לראות מה יש אחרי ה-#.
3. מנגנונים לסטיטט את ה-`route` הנוכחי.
4. השינוי בסטייטגורם לקומפוננטה להתרנדר מחדש. ברגעור בודקים את ה-`route` הנוכחי ובוחרים את הקומפוננטה.

אחרי שהבנו את הצעדים ההכרחיים – נצפה בקומפוננטה המאפשרת את זה:

```
import { useState, useEffect } from 'react';
import Home from './Home';
import Help from './Help';
import About from './About';

function Router(): JSX.Element {
  const [route, setRoute] =
    useState<string>(window.location.hash.substring(1));

  useEffect(() => {
    const handleHashChange = () => {
      setRoute(window.location.hash.substring(1));
    };
    window.addEventListener('hashchange', handleHashChange);

    return () => {
      window.removeEventListener('hashchange', handleHashChange);
    };
  }, []);

  let Child: React.ElementType;

  switch (route) {
    case '/about':
      Child = About;
      break;
    case '/help':
      Child = Help;
      break;
    default:
      Child = Home;
  }

  return (
    <div>
      <h1>App</h1>
      <ul>
        <li><a href="#/about">About</a></li>
        <li><a href="#/help">Help</a></li>
        <li><a href="#/home">Home</a></li>
```

```

        </ul>
      <Child />
    </div>
  );
}

export default Router;

```

את הקומפוננטה נציג כMOVBN ב-App באפליקציית Vite שלנו:

```

import './App.css'
import Router from './Router';

function App() {
  return (
    <>
    <Router />
    </>
  )
}

export default App

```

נסה להבין מה קורא בקומפוננטת Router. הדבר הראשון שאני עושה הוא ליצור סטיטו בשם route ולתת לו ערך ראשוני של מה שיש בכתובת #:

```

const [route, setRoute] =
useState<string>(window.location.hash.substring(1));

```

למשל, אם אני טוען את הקומפוננטה כשהאני נמצא ב:

<http://localhost:5173/#/home>

הסטיטו יהיה home/. הקוד יחזיר לי את השורה הבאה:

```
window.location.hash.substring(1)
```

הדבר השני שאני עושה הוא שאני גורם לקומפוננטה, באמצעות הhook `useEffect`, בrendsור הראשוני בלבד, להציג אירוע שישנה את ה-`route` בסטייט בהתאם למה שיש אחרי ה-#. אני מודא שמדובר בrendsור הראשוני בלבד כי אני מעביר בו-`useEffect` פרמטר שני של מערך ריק:

```
useEffect(() => {
  const handleHashChange = () => {
    setRoute(window.location.hash.substring(1));
  };

  window.addEventListener('hashchange', handleHashChange);

  return () => {
    window.removeEventListener('hashchange', handleHashChange);
  };
}, []);
```

**שימוש לב:** מפני שהפרמטר השני שמועבר ל-`useEffect` הוא מערך ריק [], אז הפונקציה שאנו מעבירו מתחילה רק בrendsור הראשוני של הקומפוננטה – ככלומר פעם אחת. זה חשוב, כי אם נציג אירוע ל-`hashchange` בכל רndsור של הקומפוננטה זו תהיה לנו זליגת זיכרון חמורה. אנו מציגים אירוע פעם אחת בלבד.

הדבר השלישי בעצם הוא לבצע בדיקה מה הסטייט הפעיל ומחליפה את משתנה Child בקומפוננטה המתאימה לסטטיויט. שימוש שאני משתמש במשתנה שמתחליל באות גודלה – child. זה משומם שמדובר במשתנה המכיל קומפוננטה ואני מצין בכך ריאקט שמדובר בקומפוננטה:

```
let Child: React.ElementType;

switch (route) {
case '/about':
  Child = About;
  break;
case '/help':
  Child = Help;
  break;
default:
  Child = Home;
}
```

**חשוב:** כדאי לשום לב שהמשתנים import Home>About, Help מגיעים מהצחרת -the

הצעד האחרון הוא לрендר את התוצאה:

```
return (
<div>
  <h1>App</h1>
  <ul>
    <li><a href="#/about">About</a></li>
    <li><a href="#/help">Help</a></li>
    <li><a href="#/home">Home</a></li>
  </ul>
  <Child />
</div>
);
```

התוצאה מתרנדרת ברכיב Child, שמשתנה בכל פעם. המשמש לווחץ על קישור, והairoו של ה- hashchange עובד ומשנה את הסטייט. שינוי הסטייט גורם לרנדור מחדש, שבו Child מקבל קומפוננטה חדשה.

כאמור, כך זה נראה באופן מאד גס ופשוט. אם נרצה שזה יעבוד עם קישורים אמיתיים ולא עם hash נצטרך להשתמש ברכיבים שונים במקצת ולהפריד, כמובן, בין הקונפיגורציה לרכיב הרואטינג (למשל לשים את כל הנתיבים בקובץ JSON או על השרת), אבל בגודל – כך זה נראה. רואוטינג בריאקט אינו וודו וכל למש אותו, אבל בעולם האמתיי כדאי להשתמש בקומפוננטה מוכנה מראש.

הקומפוננטה הפופולרית ביותר נראית, בפשטות, react-router, ואני נלמד עליה בוגר לראוטינג. אתם העקרונות המנחים אותנו רלוונטיים גם לראוטינג של כל קומפוננטה זו משתמשת ממש בכתובות אמיתיות ולא ב-#, אך העיקרון זהה. react-router היא ספריית רואוטינג כללית וספרייה נוספת שלה, react-router-dom, היא המשלימה לאפליקציות ווב. אם כן, נלמד להשתמש בשתייה.

ראשית, ניכנס לדף של react-router

<https://github.com/ReactTraining/react-router>

אפשר לראות שיש שם מדריך מסודר וdockumentציה טובה. זה כלל מפתח בוגע לכל קומפוננטה שאתם רוצים להשתמש בה בריאקט (ובכלל בכלל מערכת) – Dockumentציה טובה היא הכרחית. אם אין Dockumentציה, עדיף לא להשתמש באוותה קומפוננטה. אבל זה לא המקרה פה.

כיוון שמדובר בקומפוננטה חיצונית, אנו נתקין אותה. ההתקנה נעשית, בדיק כmo כל קומפוננטה חיצונית אחרת, באמצעות `npm install react-router-dom` למדנו בקורס פרק על קומפוננטות חיצונית. ננכדים עם הטרמינל אל המיקום של הפרויקט שלנו ומתקינים את קומפוננטת react-router ואת router-dom באמצעות:

`npm install react-router-dom`

גם כאן, על מנת להציג, נשתמש בקומפוננטות `Home`, `About` ו-`Help`. בנגד דוגמה שהראינו, של רואוטינג בסיסי, `react-router-dom` עובדת עם כתובות אינטרנט אמיטיות – קלומר ללא הסימן `#`. יש לנו כמה יתרונות וIMPLEMENTATION נעשה באותה הדקה.

המודולוגיה זהה. אני מגדיר את הרואTING ואיזו קומפוננטה נטענת בעקבותיו:

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";
import Home from "./Home";
import Help from "./Help";
import About from "./About";

function AppRouter(): JSX.Element {
  return (
    <Router>
      <div>
        <h1>App</h1>
        <ul>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/help">Help</Link>
          </li>
          <li>
            <Link to="/">Home</Link>
          </li>
        </ul>

        <Routes>
          <Route path="/about" element={<About />} />
          <Route path="/help" element={<Help />} />
          <Route path="/" element={<Home />} />
        </Routes>
      </div>
    </Router>
  );
}

export default AppRouter;
```

אם עברתם על תת-הפרק שסביר איך מבצעים את הרואTING ללא מודול חיצוני, זה יכול להירותו לכם פשוט עד כדי גיחוך. מאחורי הקומפוננטה יש שתי תפיסות עיקריות:

1. כל קישור לנútב נעשה באמצעות קומפוננטת `Link`.
2. הקומפוננטה הראשית היא ראותר ו-`switch` ממומש לא ב-`switch` רגיל אלא בעזרת `קומפוננטה`.

זה הכל. ברגע שambilים איך ראותינג עובד באופן טבעי, גם הקומפוננטות של הראותינג נראות פשוטות.

כל מואוד להשתמש גם בפרמטרים של URL. בדוקו מנטציה מסוימת אינן יכולה לעשות זאת. כל מה שעליינו לעשות הוא להוסיף את הפרמטר עם נקודותים בקומפוננטת Route שנמצאת בתוך ה-`:Routes`

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";
import Home from "./Home";
import Help from "./Help";
import About from "./About";

function AppRouter(): JSX.Element {
  return (
    <Router>
      <div>
        <h1>App</h1>
        <ul>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/help/:id">Help</Link>
          </li>
          <li>
            <Link to="/">Home</Link>
          </li>
        </ul>

        <Routes>
          <Route path="/about" element={<About />} />
          <Route path="/help/:id" element={<Help />} />
          <Route path="/" element={<Home />} />
        </Routes>
      </div>
    </Router>
  );
}

export default AppRouter;
```

השינוי היחיד שבוצע מהראוטינג הבסיסי יותר הוא:

```
<Route path="help/:id" element={<Help />} />
```

זה בעצם מאפשר לנו להכניס פרמטרים, למשל:

<http://localhost:5173/help/6382020>

והפרמטר הזה יהיה זמין לנו בקומפוננטת Help באמצעות useParams, שהוא פונקציה ש מגיעה עם react-router. למשל כך:

```
import { useParams } from "react-router-dom";

function Help(): JSX.Element {
  const { id } = useParams<{ id: string }>();

  return (
    <div>
      <h1>Help</h1>
      <p>ID: {id}</p>
    </div>
  );
}

export default Help;
```

נסו את זה בעצמכם. הציבו את הקוד הזה ב-Vite שלכם וראו עד כמה זה פשוט. בדוקומנטציה של react-router יש דוגמאות רבות לשימוש שכדי לבחון. אבל כאמור, אם הבנתם איך ממשים ריאקט你自己 בעצמכם ואיך העיקרון עובד, לא יהיה לכם קשה להבין ולהשתמש בקומפוננטות של ניוט.

**תרגום:**

צרו שני קומפוננטות: Home, שזהה לקומפוננטה מתחילה הפרק, והקומפוננטה הזו, המציגת תמונה אקרואית מהרשות:

```
function NiceImage(): JSX.Element {
  let imgSrc:string = 'https://picsum.photos/id/237/200/300';
  return (
    <img src={imgSrc} />
  );
}

export default NiceImage;
```

צרו אפליקציית ריאקט שמשתמשה בראוטינג. באמצעות הראוטינג אפשר להחליף בין Home לבין NiceImage. אל תשתמשו בראוטינג של קומפוננטה חיצונית.

**פתרון:**

```
import { useState, useEffect } from 'react';
import Home from './Home';
import NiceImage from './NiceImage';

function Router(): JSX.Element {
  const [route, setRoute] =
  useState<string>(window.location.hash.substring(1));

  useEffect(() => {
    const handleHashChange = () => {
      setRoute(window.location.hash.substring(1));
    };
    window.addEventListener('hashchange', handleHashChange);

    return () => {
      window.removeEventListener('hashchange', handleHashChange);
    };
  });
}
```

```
}, []);  
  
let Child: React.ElementType;  
  
switch (route) {  
  case '/niceimage':  
    Child = NiceImage;  
    break;  
  default:  
    Child = Home;  
}  
  
return (  
  <div>  
    <h1>App</h1>  
    <ul>  
      <li><a href="#/niceimage">Nice Image</a></li>  
      <li><a href="#/home">Home</a></li>  
    </ul>  
    <Child />  
  </div>  
);  
}  
  
export default Router;
```

זה שינוי קל מהדוגמה שהוצגה בפרק עצמו. מה שחשוב לראות הוא איך ה-`switch case`, שמוסע בכל פעם שה-`hash` (זה ה-`#`) ב-URL משתנה, מחליף את משתנה ה-`route` בהתאם לערך של מה שיש אחרי ה-`hash` וטוען קומפוננטה אחרת.

**תרגיל:**

ממשו את הפתרון הקודם באמצעות `.react-router`

**פתרונות:**

```
import { Routes, Route, Link, Outlet, BrowserRouter } from "react-router-dom";
import Home from "./Home";
import NiceImage from './NiceImage';

function Layout(): JSX.Element {
  return (
    <div>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/nice-image">Nice Image</Link>
          </li>
        </ul>
      </nav>
      <Outlet />
    </div>
  );
}

function MyRouter(): JSX.Element {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="nice-image" element={<NiceImage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

export default MyRouter;
```

כיוון שבמדובר בראוטינג פשוט, אני נדרש לעשות את הצעדים הבאים באפליקציה שלי:

1. התקנת react-router-dom באמצעות:  
`.npm install react-router-dom`
2. יצירת רכיב שנקרא MyRouter (או כל שם אחר).
3. ביצוע `import` לכל הקומפוננטות שאני צריך.
4. עדיפות כל הקומפוננטה בקומפוננטה ראשית מסוג `BrowserRouter`.
5. תפריט וקישורים באמצעות רכיב `Link`:  
`<Link to="/nice-image">Nice Image</Link>`
6. יצירת קומponentת `Layout` במקום אמורות להיות הקומפוננטות שאני מגיע אליהן בינווות.
7. להציב בתוך קומponentת `Routes` את הקומפוננטות שנטענו כתוצאה מפעולת הראוטינג.  
למשל:  
`<Route path="/image"><NiceImage /></Route>`

**תרגיל:**

בפתרון התרגיל הקודם, אפשר להחליף את התמונה באמצעות שינוי מספר ה-`p0` באופן הבא:

`https://picsum.photos/id/${id}/200/300`

כלומר ה-`p0` יכול להיות:

`https://picsum.photos/id/100/200/300`

או:

`https://picsum.photos/id/44/200/300`

וכך הלאה.

הכניסו ניוט באמצעות פרמטרים, כך שתוכלו להכניס מספר ב-URL, למשל:

`http://localhost:5173/nice-image/100`

או:

`http://localhost:5173/nice-image/44`

ותמונה עם ה-`p0` המתאים תיתען.

**פתרונות:**

קומפוננטת הראוטינג:

```

import { Routes, Route, Link, Outlet, BrowserRouter } from "react-router-dom";
import Home from "./Home";
import NiceImage from './NiceImage';

function Layout(): JSX.Element {
  return (
    <div>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/nice-image/200">Nice Image 200</Link>
          </li>
          <li>
            <Link to="/nice-image/44">Nice Image 44</Link>
          </li>
        </ul>
      </nav>
      <Outlet />
    </div>
  );
}

function MyRouter(): JSX.Element {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="nice-image/:id" element={<NiceImage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

export default MyRouter;

```

## קומפוננטת :NiceImage

```

import {
  useParams
} from 'react-router-dom';

function NiceImage():JSX.Element {
  let params = useParams();
  let imgSrc:string =
`https://picsum.photos/id/${params.id}/200/300`;
  return (
    <img src={imgSrc} />
  );
}
export default NiceImage;

```

בראוטיניג הפתרון הוא פשוט ביותר – הוספת הפרמטר `id` ב-`path` בקומפוננטת `Route`.  
 בקומפוננטה שאמורה לRAPPORT את ה-`id`, הלווא היא `NiceImage`, צריך לבצע `import {useParams}` שמנגינה אליו מ-`react-router`, לקבל את ה-`id` ולשתול אותו ב-`imgSrc`.  
 אפשר למשגם שדה או תיבת `Select` שבהם כתובים מספר, והתמונה המתאימה תיתען.

פרק 20

# הונתקם



# קונטקסט

קונטקסט הוא מונח חשוב בריект. הוא מאפשר לנו לחלק מידע בין קומפוננטות שונות, בדומה לסטיטית, אבל הוא גלובלי. ביום אנו מכירים שתי דרכים להעברת מידע בקומפוננטות: האחת היא באמצעות `props`, כאשר קומפוננטה האב מעבירה לקומפוננטה הבנות מידע דרך ה-`props`, והשנייה היא אירועים – קומפוננטה הבת מפעילה אירוע כתוצאה מפעולה כלשהי (משתמש או טיימר). אם קומפוננטה האב העבירה פונקציה שmaps פעולה כתוצאה מאירוע כלשהו (`Handler`), הוא יופעל כאשר אירוע מופעל בקומפוננטה הבת.

הבעיה היא שזה יכול להסתנן. לעיתים קומפוננטות רבות מעצם היררכיות שונות צריכות לחלק מידע, כמו למשל סטיילינג (עיצוב) – מידע על עיצוב כמו גודל פונט, פלטת צבעים וגדלים, או מידע על משתמש. קומפוננטה מקבלת מצד השרת את המידע על המשתמש וצריכה להעביר את המידע הזה הלאה. לעיתים יש מידע בקומפוננטה אב מסוימת וצריך להעביר אותו לקומפוננטה שנמצאת חמיש רמות מתחתיה בהיררכיה ולשרר את המידע. זה מורכב. קונטקסט מאפשר לנו לשומר מגיר מידע גדול שממנו כל קומפוננטה יכולה לקרוא, מעין סטייט גלובלי.

בקונטקסט יש שני משתתפים: האחד הוא הפרוביידר (`Provider`) והשני הוא הלקוח (`Consumer`). הפרוביידר חייב תמיד לעסוק את הלקוח ולהיות גבוהה יותר ממנו בהיררכיה, אחרת הלקוח לא יוכל לגשת לקונטקסט שלו התו הפלביידר מספק. הפרוביידר יכול לאכלס את הקונטקסט באופן סטטי או לבצע קריאה ל-`API` חיצוני. הלקוח מקבל גישה לקונטקסט באמצעות הוקם.

פרוביידר לא חייב להיות קומפוננטה ודרך כלל לא בניו קומפוננטה אלא קובץ טיפוסKrøppet רגול, למשל UserContext.ts, שגדיר קונטקסט של משתמש:

```
import React from 'react';

export type UserInfo = {
  data: {
    name: string;
    surName: string;
  };
  changeUser: Function;
};

const initialValue: UserInfo = {
  data: {
    surName: "",
    name: "",
  },
  changeUser: () => {},
};

const UserContext = React.createContext<UserInfo>(initialValue);
export const UserProvider = UserContext.Provider;
export const UserConsumer = UserContext.Consumer;
export default UserContext;
```

כדי לשים לב שאין לנו פה קומפוננטה, אלא יצירת קונטקסט באמצעות:

```
const UserContext = React.createContext<UserInfo>(initialValue);
```

האובייקט `initialValue`, שמועבר כפרמטר ל-`createContext`, הוא בעצם בירית מיוחד – ערך שמוחזר רק כאשר קומפוננטה מבצעת קריאה לפרוביידר זהה, אם הוא לא משוויך אליה. שימושו לב לטייפ פה – אני משתמש בטיפ `UserInfo` ומעביר אותו באופן גנרי ל-`createContext`.

במקרה הזה אני מגדיר את `UserInfo` בתוך הקובץ עצמו כדי לפשט את הדוגמה. במקרה אני אגדיר את סוג המידע הזה בקובץ נפרד וاعשה לו `export` כי אני צריך להשתמש בו בעוד שני מקומות – בקומפוננטה הצורכת את הקונטקסט.

אני משתמש ב-API של קונטקסט כדי ליצור וליצא את הפרוביידר, את הצרךן ואת הקונטקסט. אני יכול לבחור לפרוביידר, לצרcn או לkontexst תחילה כרצוני. במקרה זה, כיוון שמדובר בkontext של משתמש, בחרתי בתחילה של `User`.

אחרי שהגדרתי קונטקסט, אני יכול להשתמש בפרופיבידר. הפרוביידר מספק לכל הקומפוננטות שutowפות אותו את הקונטקסט שהוא נותן. אני יכול לשים אותו בכל מקום, אפילו ב-app. הנה דוגמה:

```
import './App.css';
import { UserProvider } from './userContext'
import Welcome from './Welcome';

type userInfo = {
  name: string,
  surName: string,
  city: string,
}

function App():JSX.Element {
  const user:userInfo = {
    name: 'Ran',
    surName: 'Bar-Zik',
    city: 'Petah-Tiqwa'
  };
  return (
    <div className="App">
      <header className="App-header">
        <UserProvider value={user}>
          <Welcome />
        </UserProvider>
      </header>
    </div>
  );
}
export default App;
```

ראשית אני מיבא את הפרוביידר מהקונטקסט שיצרתי. כזכור, אני מיצא שם פרוביידר, צרכן וקונטקסט.

```
import { UserProvider } from './UserContext'
```

אני בוחר את המידע שאני רוצה להעביר בקונטקסט. המידע יכול להיות סטטי או מידע ש מגיע משירות (כלומר מהשרת). בדוגמה זו אני משתמש במידע סטטי:

```
const user:userInfo = {
  name: 'Ran',
  surName: 'Bar-Zik',
  city: 'Petah-Tiqwa'
};
```

שים לב שאני חייב להגיד כאן סוג מידע בשם userInfo או בכלל שם אחר, אבל אני לא יכול להגיד משתנים בלי סוג מידע. אנו עדין בטיפוס קרייפט

הצעד האחרון הוא להשתמש בפרופויבידר כמו בקומפוננטה רגילה ולהעביר אליו את המידע. כל מה שיהיה עטופ בקומפוננטה זו וgemäßון כל הקומפוננטות שמתוחת לקומפוננטה העטופה, ככלומר אלו שתחתיה בהיררכיה, יקבלו גישה לקונטקסט:

```
<UserProvider value={user}>
  Any component
</UserProvider>
```

עכשו במקום Any Component נשים קומפוננטה אמיתית, למשל `:Welcome.tsx`

```
<UserProvider value={user}>
  <Welcome />
</UserProvider>
```

לא נשכח לעשות `import`, כמובן, לפני שאנו מציבים אותה.

איך הקומפוננטה זו תיראה?

```
import { useContext } from 'react'
import UserContext from './UserContext'

type userInfo = {
  name: string,
  surName: string
}

function Welcome(): JSX.Element {
  const user:userInfo = useContext<userInfo>(UserContext);
  return (
    <span>Hello, {user.name} {user.surName}!</span>
  );
}
export default Welcome;
```

הקומפוננטה זו משתמשת בקונטקסט באמצעות הוק. השלב הראשון הוא להביא הוק `useContext`, שעובד בדיקות כמו כל הוק אחר:

```
import { useContext } from 'react'
```

השלב השני הוא להביא את הקונטקסט שאנו מעוניינים להשתמש בו. במקרה זהה, `:UserContext`

```
import UserContext from './UserContext'
```

השלב הבא והאחרון הוא פשוט לצורך את המידע שהפרוביידר הזה מביא לנו, ללא טייפסקרייפט, הפונקציה הזו נראה כך:

```
const user = useContext(UserContext);
```

אבל עם טייפסקרייפט זה נראה כך, זה מאיים בהתחלה אבל נפרק את זה:

```
const user:userInfo = useContext<userInfo>(UserContext);
```

ראשית הגדרת הקבוע עם סוג המידע `userInfo`. את זה אנו אמורים להזכיר:

```
const user(userInfo
```

השלב השני הוא העברת סוג המידע הזה ל `userContext` באמצעות גנריות. כלומר אנו מגדירים ל `userContext` את סוג המידע שהוא צריך להחזיר.

```
(UserContext as React.Context<userInfo>);
```

אנו חייבים להגדיר את הסוג הזה כ `ReactContext`. אבל `ReactContext` זה לא סוג מידע רגיל. זה סוג מידע שבעצמו מקבל את ההגדרה באמצעות גנריות ואני נדרש להעביר לו את סוג המידע `.userInfo`.

מהשלב הזה, כל מה שיש ב-`user` הוא מה שיש לנו בפרופוביידר.

אם הפורוביידר מתעדכן מסיבה מסוימת – הוא ירנדר מיד את הקומפוננטות שימושות בו. שימושו לב שהקומפוננטה צריכה להיות מודעת ל-`UserContext` וולעשות לו `import`.

אבל מה קורה אם אנו רוצים לעדכן את הקונטקסט ממוקם אחר שהוא לא הפורוביידר? אם אנו רוצים לחת לפורוביידר אפשרות להתעדכן על ידי אחת מהקומפוננטות בתחום ההיררכיה, אנו צריכים להעביר, נוספת על הקונטקסט, גם פונקציה שתשנה אותו.

אני אדגים זאת באמצעות הוספת כפטור טיענית משתמש בקומפוננטת `Welcome` שটוען משתמש חדש.

הצעד הראשון צריך לעשות הוא ליצור סטייט בפונקציה המכילה את הפורוביידר. במקרה זהה `App.tsx`. בסטייט זהה אני מכניס את הערכים הראשוניים שלי ופונקציה המשנה את אותו סטייט. לפורוביידר אני כבר לא מכניס את אובייקט המידע בלבד אלא גם את אובייקט המידע תחת תכונת `data` וגם את הפונקציה המשנה את הסטייט. אני יכול לקרוא לפונקציה המשנה את הסטייט `changeUser`. באיזה שם שאני רוצה. כאן בחרתי ב `changeUser`.

```
import './App.css';
import { useState } from "react";
import { UserProvider } from './userContext'
import Welcome from './Welcome';

type UserInfo = {
  name: string,
  surName: string,
}

function App():JSX.Element {

  const [user, setUser] = useState({
    name: "Ran",
    surName: "Bar-Zik",
  });
  const providerOptions = {
    data: user,
    changeUser: (value: UserInfo) => setUser(value),
  };
}
```

```

return (
  <div className="App">
    <header className="App-header">
      <UserProvider value={providerOptions}>
        <Welcome />
      </UserProvider>
    </header>
  </div>
);
}
export default App;

```

אפשר לראות שבמקום להכניס רק את ערך אובייקט המשמש, אני מכניס את ערך אובייקט המשמש בסטייט ומעביר אובייקט שיש בו גם את הסטייט וגם פונקציה שמשנה אותו.

בפרוביידר נמצא `data` ושם הפונקציה המשנה אותו. אני צריך אותם באמצעות  `useContext`.

אם `Welcome.tsx` רוצה לשנות את המידע הזה, הוא צריך להפעיל את הפונקציה שמשנה את הסטייט שבו נמצא הפרוביידר, וכך הוא ישנה אותו. כך פונקציית `Welcome` נראית.

```

import { useContext } from "react";
import UserContext from "./userContext";
import Button from "@mui/material/Button";

type UserInfo = {
  name: string;
  surName: string;
};

function Welcome(): JSX.Element {
  const { data, changeUser } = useContext(UserContext);

  const newUser: UserInfo = {
    name: "Moshe",
    surName: "Cohen",
  };

  function clickHandler(): void {

```

```

        changeUser(newUser);
    }

    return (
        <div>
            <span>
                Hello, {data.name} {data.surName}!
            </span>
            <Button variant="contained" color="primary"
onClick={clickHandler}>
                Load another user
            </Button>
        </div>
    );
}

export default Welcome;

```

בקומפוננטת `Welcome.tsx`, ראשית אני שואב את אובייקט המשתמש (כדי שאוכל להציג אותו) וכן את הפונקציה שמשנה את הסטייט של הפרוביידר. את הפונקציה אני מפעיל באירוע הלחיצה של הכפתור.

از בעצם מה שקרה הוא:

1. אני מגדר קונטקסט בקובץ נפרד.
2. בוחר היכן להציב את הפרוביידר שלי, בדרך כלל בקומפוננטת אב. במקום שבו אני מציב את הפרוביידר, אני מזמין ערך. הדרך הכי טובה לעשות את זה היא באמצעות סטייט, אם כי זו לא חובה.
3. אני יכול להגיד ייחד עם הערך פונקציה שמשנה את הסטייט של הפרוביידר. כל מי שמשתמש בكونטקסט זה יכול (אם הוא רוצה) להשתמש בה ולשנות את הקונטקסט בכלל.

קונטקסט הוא דרך מצוינת לחלק מיידע בין קומפוננטות שונות וגם קל להשתמש בו עם הוקים. הדרך לעדכן קונטקסט יכולה להירות מסורבלת, אבל אם מבינים שברגע שיוצרים קונטקסט ומידע גם יוצרים פונקציה חדשה אותו ומעבירים אותה – אז הכל נהייה פשוט.

**תרגיל:**

העתיקו את קומפוננטת Welcome.tsx וקומפוננטת App.tsx ל-Vite App.

הוסיפו את שם העיר לאובייקט המידע. הציגו את שם העיר בקומפוננטה נפרדת בשם Welcome CityName.tsx שמציגה את שם העיר של המשתמש. הקומפוננטה תוצב בקומפוננטת Welcome.tsx ותצרוך שירות מהפרוביידר.

**פתרונות:**

ראשית, ניצור את הקומפוננטה. הקומפוננטה היא קומפוננטה שאמורה לצורך קונטקסט ונראית ככזה:

```
import { useContext } from "react";
import UserContext from "./UserContext";

type UserInfo = {
  name: string;
  surName: string;
  city: string;
};

function CityName(): JSX.Element {
  const user: UserInfo = useContext<UserInfo>(UserContext);
  return <span>You are from {user.city} </span>;
}

export default CityName;
```

השורות המעניינות מבחןתנו הן:

```
import { useContext } from 'react'
import UserContext from "./UserContext";
```

כעת ניבא את ההוק :useContext

הקומפוננטה זו דומה לקומפוננטה `Welcome.tsx`.

ומה קורה בקומפוננטת Welcome? אין חדש. אנחנו רק צריכים להציג את קומפוננטת CityName.tsx כדי להציג את שם העיר.

```
import { useContext } from 'react'
import UserContext from './UserContext'
import CityName from './CityName';

type UserInfo = {
  name: string,
  surName: string,
  city: string,
}

function Welcome(): JSX.Element {
  const user: UserInfo = useContext<UserInfo>
    (UserContext);
  return (
    <div>
      <span>Hello, {user.name} {user.surName}!</span>
      <span><CityName /></span>
    </div>
  );
}
export default Welcome;
```

האם צריך לקשר את קומפוננטת CityName לפורמיידר? לא. כל עוד היא בת, נצדה, נינה או הבת של הנינה של ה-Provider זה לא משנה – היא יכולה להשתמש בקונטקסט.

פרק 21

# חיבור לשורת נס סרווייסים



## חיבור לשרת עם שירותי

פרק זה מניח ידוע מוקדם ב-AJAX ובהבדל בין הבקשות השונות – לפחות ברמת פונקציית ה-`fetch` הטבעית.

חיבור לצד שרת הוא קרייטי בכל הנוגע לאתר או לאפליקציה המבוססים על ריאקט (או בכלל). רוב האתרים אינם אתרים סטטיים אלא אתרים המתעדכנים במידע המגיע מהשרת. המידע המגיע מהשרת נכנס כבר כ-`props` לקומפוננטות מקומפוננטות אב כלשהן או לكونטקסט. את הקראיות אלו מבצעים באמצעות AJAX ואפשר להשתמש ב-`fetch` שmagiu עם ג'אווהסקריפט או בכל ספרייה אחרת, כמו הספרייה הנפוצה `Axios`.

כשאנו מרכיבים קוד בריאקט, הוא רץ בדף בלבד. נכון, `Vite` מריםה שרת פיתוח במחשב שלנו לשם הנוחות. אבל בסופו של יום, בסביבת הפודקשן, הסביבה האמיתית, השרת יכול להיות שונה לחולtin. הדף יכול לקבל אתקובצי הג'אווהסקריפט שמורכבים מספריית ריאקט ומהקוד שלכם מכל שרת שהוא. אחרי שדף הלקוח מורייד אתקובצי הג'אווהסקריפט האלו, הוא מרים את האפליקציה הבנויה על ידי ריאקט על הדף בלבד. אם הוא ציריך מידע מהשרת, הקוד שלכם אמרור להביא לו אותו.

בדרך כלל, מתכווני צד השרת כתובים API בצד השרת כדי שיחזר לכם מידע, והאחריות של מתכווני ריאקט היא לשולח בקשות לשרת. כן, למשל, אתם משתמשים בראוטינג דף התחברות עם שם משתמש וסיסמה. המשתמש מקליד שם משתמש וסיסמה ולוחץ על כפתור "שלח". האחריות שלכם משמשת וסיסמה. המשתמש מקליד שם משתמש וסיסמה ל-API של צד השרת. צד השרת שלכם היא לשולח, באמצעות AJAX, את שם המשתמש וסיסמה ל-API של צד השרת. צד השרת אומר להחזיר לכם תשובה של 200 ועובדיקט המכיל את פרטי המשתמש, בהנחה שהcoil מצלי, ולשתול עוגייה בצד המשתמש. האחריות שלכם היא לחת את המידע הזה ולהכניס אותו לكونטקסט כדי שכל המידע באתר יוצג למשתמש המחבר כמו שציריך.

כמו בראוטינג, ריאקט אינה קובעת עבור המשתמש איך לשולח את הבקשות ואין לנול אותן. אפשר לבצע את הבקשות בעזרת `fetch` הטבעי שיש בג'אווהסקריפט בסביבת הדף או בעזרת ספרייה. הספרייה הפופולרית לניהול בקשות AJAX היא `Axios`. אנו נתרגל באמצעות `fetch`.

יש כמה דרכים לנצל את הבקשות – אפשר לכתוב אותן בתוך הקומפוננטה ממש. אנו נתאמן עם ה-API של בדיחות צ'אק נוריס, שבעצם מה אפשר לנו לקבל בבדיקה צ'אק נוריס בקישור הבא:

<https://api.chucknorris.io/jokes/random>

נסו בעצמכם! העתיקו והדביקו את הקישור בדף וצפו ב-JSON שmagיע.  
איזה נमש את זה? כמובן – אפשר בקומפוננטה:

```
import { useState, useEffect } from 'react'

function Welcome(): JSX.Element {
  const [joke, setJoke] = useState<string>('Joke is loading...');

  function getJoke(): void {
    fetch('https://api.chucknorris.io/jokes/random', {})
      .then((response) => {
        return response.json();
      })
      .then((jsonObject) => {
        setJoke(jsonObject.value);
      });
  }

  useEffect(getJoke, []);

  return (
    <div>
      {joke}
    </div>
  );
}

export default Welcome;
```

אנו מכנים את קריאת ה-AJAX, באמצעות `fetch`, אל פונקציה מסודרת שאנו מפעילים רק פעמי'ת, באמצעות ההוק `useEffect` שמקבל כפרמטר ראשון את הפונקציה להפעלה וכפרמטר שני מערך ריק, שבעצמו אומר שהוא יופעל רק כאשר הקומפוננטה מורנדרת. הפונקציה שעושה גם מכינה את הבדיקה המתבקשת אל הסטייט.

הקוד הזה, שאמור להיות מובן מאוד אם אתם כבר מכירם AJAX וידעים איך `fetch` עובד, הוא קוד ולידי, אבל הוא רעיון גrosso מאד אם האפליקציה שלכם מעת יותר מורכבת, למשל אם יש דרישות נוספות כמו להכניס את תוכאות הבדיקה לקונטנסט, כדי שעוד קומפוננטות יוכל להציג אותה, או לשנות את ה-API. זו הסיבה שמקובל מאוד להכניס קריאות לשרת לפונקציות ג'אויסקייפט טהורות שנקראות שירותים (Services). יוצרים בפרויקט תיקיה שנקראת `Services` ושם שמים את כל הפונקציות המטפלות בקריאות לשרתים.

למשל, אם אני רוצה להמיר את הקריאה לקומפוננטה זו בקריאה לשירות אמיתי, השירות יראה כך:

```
export default function getChukJoke():Promise<string> {
  return fetch(`https://api.chucknorris.io/jokes/random`, {})
    .then((response) => {
      return response.json();
    })
    .then((jsonObject) => {
      return jsonObject.value;
    });
}
```

כשארצה להשתמש בו, פשוט אבצע import לשירות זהה.

```
import { useState, useEffect } from 'react'
import getChukJoke from './services/chuckJokes';

function Welcome(): JSX.Element {
  const [joke, setJoke] = useState<string> ('Joke is loading...');

  useEffect(getJoke, []);

  function getJoke():void {
    getChukJoke().then((joke) => {setJoke(joke);});
  }

  return (
    <div>
      {joke}
    </div>
  );
}

export default Welcome;
```

יש שירותים שנבנו כקלאסים ויש כאלה שנבנו כפונקציות. כל דרך היא טובה – אבל כדאי לשום לב שלא משנהאיזה דרך נבחרה – מדובר בטיפוסקריפט טהור. ריאקט לא מתערבת בעניינו השירותים וכל אחד יוכל למשרווותם לפי הבנתו.

סביר להניח שכאשר תעבדו מול מערכות אמיתיות, מי שיגדר לכם איך השירותים מתנהלים מול השרת יהיו מתכנתיו הבק אנד, והם אלו שיגדרו לכם איך להתחבר.

**תרגיל:**

נדרשתם להציג בדיחת אבא רנדומלית באתר. מתכונת צד השרת הגדר לכם API באופן הבא: יש לשלח בבקשת GET אל הכתובת:  
<https://icanhazdadjoke.com>

עם בקשה ה-GET דואגים לשלח את ה-`header: {Accept: 'application/json'}` כדי לקבל את בדיחת האבא.

**רמז:** בבקשת `fetch` עושים באופן הבא:

```
return fetch(`https://icanhazdadjoke.com`, {
  headers: {
    'Accept': 'application/json'
  },
})
```

**פתרונות:**

```
export default function getDadJoke():Promise<string>{
  return fetch(`https://icanhazdadjoke.com`, {
    headers: {
      'Accept': 'application/json'
    },
  })
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    return jsonObject.joke;
  });
}
```

משתמשים ב-service זהה כה:

```
import { useState, useEffect } from 'react'
import getDadJoke from './services/dadJokes';

function DadJoke(): JSX.Element {
  const [joke, setJoke] = useState<string>('Joke is loading...');

  useEffect(getJoke, []);

  function getJoke(): void {
    getDadJoke().then((joke) => {setJoke(joke);});
  }

  return (
    <div>
      {joke}
    </div>
  );
}

export default DadJoke;
```

בגدول, זה לא ריאקט אלא ג'אוوهסקריפט. הסרוייס הוא פונקציה שמייצרת בקשהAJAX. הבקשת NSLJSON. אנו לוקחים את האובייקט ומחזירים את ה-*joke* מתוכו.

הריект פשוט קורא לפונקציה שמחזירה פרומיס וכאשר הוא מתמלא, אנו מכניסים את מה שהוא נותן לנו לסטיטו של *joke*.

בשלב זה אנו יכולים לטפל, כמוון, בשגיאות או בפלט לא צפוי. למשל, משהו בסגנון זהה:

```
import { useState, useEffect } from 'react'
import getDadJoke from './services/dadJokes';

function DadJoke(): JSX.Element {
  const [joke, setJoke] = useState<string>('Joke is loading...');

  useEffect(getJoke, []);

  function getJoke(): void {
    getDadJoke()
      .then(
        (joke) => { setJoke(joke); },
        () => { setJoke('Sorry! Error!'); }
      );
  }

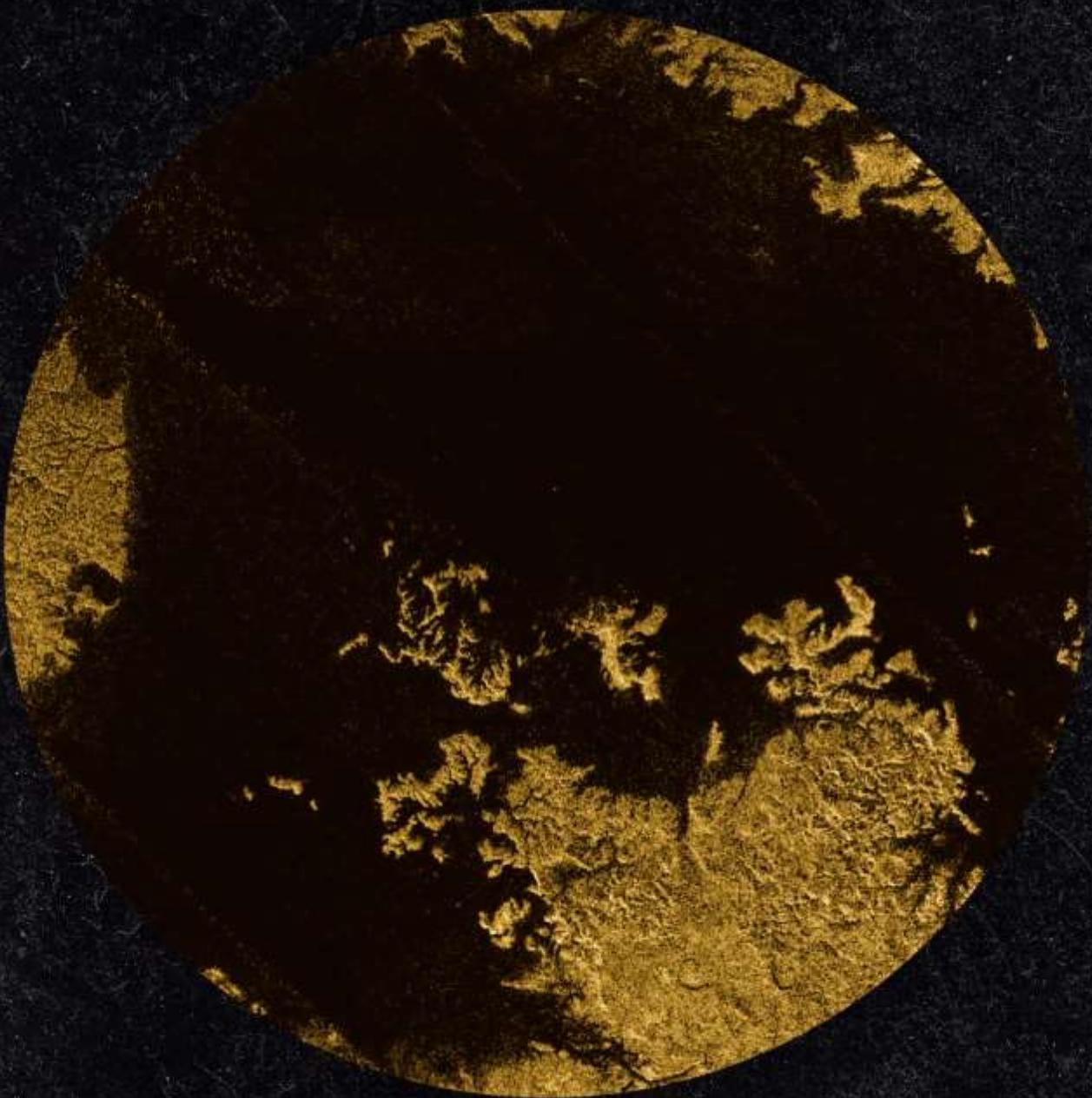
  return (
    <div>
      {joke}
    </div>
  );
}

export default DadJoke;
```

או כמוון ניהול מסודר יותר. חשוב מאד, בקריאה AJAX, לחשב על מה שקרה אם השירות לא מחזיר את התשובה המczופה.

פרק 22

# מבחן לבדיקות



# מבוא לבדיקות עם Vitest

בדיקות אוטומטיות הן חלק חשוב מפיתוח מקצועי בריект. בפיתוח יש לנו כמה סוגי בדיקות: בדיקות ייחודית (הבודקות את הקומפוננטה), בדיקות פונקציונליות (כדי לראות איך הקומפוננטה עובדת), בדיקות אינטגרציה (הבודקות איך הקומפוננטות השונות מתקשרות עם צד השרת) ובבדיקות End to End (E2E) שבודקות את כל האפליקציה מקצה למקצה.

בדיקות חיוניות לפיתוח נכון, וברוב החברות הטובות נהוג כתוב בבדיקות ייחודית. הבדיקות הללו קריטיות כיון שאם מתכנת אחר יעשה שינוי בקומפוננטה שלכם, הבדיקה תישבר והוא יהיה שיש בעיה עם תסריט מסוים שבניתם עליו ובדקתם אותו. זו הסיבה שבבדיקות הן דבר חשוב ולא מיועד רק לאנשי QA. בעולם הפיתוח מקובל שמפתחים כתובים בבדיקות ייחודית.

בדיקות אוטומטיות הן סקריפטים, הכתובים בג'אוויסקריפט, ויש לכל בדיקה קriterionים להצלחה. למשל: "אם לווחצים על כפתור, האפליקציה קוראת לשירות". הבדיקות הללו רצות באופן נפרד באמצעות פקודה מסוימת ומריצים אותן לפני כל כניסה קוד חדש. בכלל זה בדיקות אוטומטיות הן חלק מהשנה `Continuous Integration`, שהוא תהליך שקוד חדש עבר כשהוא משולב בקוד שנכתב קודם לכן. אם מישחו כתוב תוספת לקוד שלנו (למשל הוסיף `props` נוספים) ויש לנו בבדיקות טבות שרצת על הקוד החדש וכל הבדיקות רצות ואף אחת מהן לא נכשלת – אנו יודעים שכנהראה נכתב קוד איקוני (או לפחות צזה שעבר את הבדיקות) ונונכל להעלות אותו לאוויר.

ישנן כמה סדריות לבדיקות שפועלות היטב עם ריאקט ועם Vite. שתיהם מהן פופולריות מאוד. הראשונה נקראת `jest` והשנייה `vitest`. הן עובדות באופן דומה למדי אך כיון שגם משתמשים ב-`Vite`, אנו נכתב ב-`vitest`.

בדיקות מקובל שקובץ הבדיקות נושא את השם של הקובץ המקורי עם התוספה `spec` או `test`. למשל, קובץ הבדיקות של `axs` יהיה `Welcome.spec.js`, אבל יש קונבנציות נוספות.

לא חייבים לכתוב את הבדיקות בטיפוסקריפט וזה נחשב במקרה מהפתרונות כעדף תקורה ונינתן לנכתב את הבדיקות בג'אוויסקריפט בלבד.

הבה נכתוב בדוקה אוטומטית ראיונה עברו `xs.ts`, שתירהה כך:

```
function Welcome(): JSX.Element {
  return (
    <p>Hello world!</p>
  );
}

export default Welcome;
```

אפשר לראות שזו קומפוננטה פשוטה מאוד. יש לה רק תפקיד אחד, להציג `Hello world!`. איך נבדק אותה?

## התקנת vitest

על מנת להשתמש ב-`vitest`, אנו נדרש להתקין אותה. ההתקנה מתקינה את `vitest` עצמה וכן סדרירות הנדרשת להפעלה שלה עם ריאקט כתוצאה שהוא לא דפדן.

ראשית יש להתקין את החבילות של `vitest`, `jsdom` וחבילה שנקראת `testing-library`.

```
npm install @testing-library/jest-dom@6.4.2 @testing-library/react@14.2.2 vitest@1.4.0 jsdom@24.0.0 -D
```

אנו מתקינים אותן במספר גרסאות כדי להקל על הלימוד.

השלב הבא הוא לאתר את קובץ `vitest.config.ts` בתיקיה הראשית (אם הוא לא קיים יש ליצור אותו) ולהכניס לתוכו את הקוד הזה:

```
import { defineConfig } from 'vitest/config';

export default defineConfig({
  test: {
    globals: true,
    environment: 'jsdom',
  },
});
```

השלב הבא הוא לכתוב קובץ בדיקות שמאפיין את הקומפוננטה ובודח את הטקסט שלה כדי לראות שהוא שווה ל-Hello world – זה נשמע פשוט וזה באמת פשוט עבור הדוגמה שלנו. ראשית, ניצור את קובץ הבדיקה: `Welcome.spec.tsx`:

בתוכו אנו עושים `import` לרכיבים שאנו צריכים וכותבים את הבדיקה, בבדיקה אנו בודקים שיש את הטקסט Hello World. כך עושים את זה:

```
import { describe, it, expect } from 'vitest';
import { render, screen } from '@testing-library/react';
import '@testing-library/jest-dom';
import Welcome from './Welcome';

describe('Welcome component', () => {
  it('renders the correct content', () => {
    render(<Welcome />);
    expect(screen.getByText('Hello world!')).toBeInTheDocument();
  });
});
```

בואו נעבור על חלק הבדיקה.

## חלק ראשון – import

אנו מיבאים כמה מתודות חשובות – `it`, `expect` ו-`describe`, שהם חלק מבני הבדיקה ונידון עליהם במסגרת הבדיקה ואת `render` ו-`screen` שימושות לנו בבחינת הפלט. אנו מיבאים גם את `-jest-dom` וכמוון את הקומפוננטה שלנו. אלו החלקים הכי חשובים לבדיקה.

## חלק שני – כתיבת מסגרת הבדיקה

`it` הוא חלק חשוב מאוד בבדיקות והוא בא כאובייקט גלובלי מ-`vite`. הוא מקבל שני ארגומנטים: האחד הוא שם הבדיקה והשני הוא פונקציית חץ שבתוכה הבדיקה.

## חלק שלישי – הרצת הקומפוננטה

במקרה זהה מדובר בפעולה פשוטה.

```
render(<Welcome />);
```

## חלק רביעי – הבדיקה

אחרי שהקומפוננטה רונדרה, אנו יכולים לבדוק את ה-HTML שבתוכו היא נמצאת. אנו עושים את הבדיקה באמצעות פונקציית `expect`. מדובר בפונקציה שגמ היא בא עם `vite` והוא מקבלת ביטוי שאליו יש לשרשר את ההנחה. במקרה זה הביטוי הוא מציאת הטקסט `Hello world` וההנחה היא `to be in the document` – כלומר "הוא יהיה במסמך" כל בדיקה היא בעצם ביטוי והנחה בונגעו אליו. ההנחה היא מתודה.

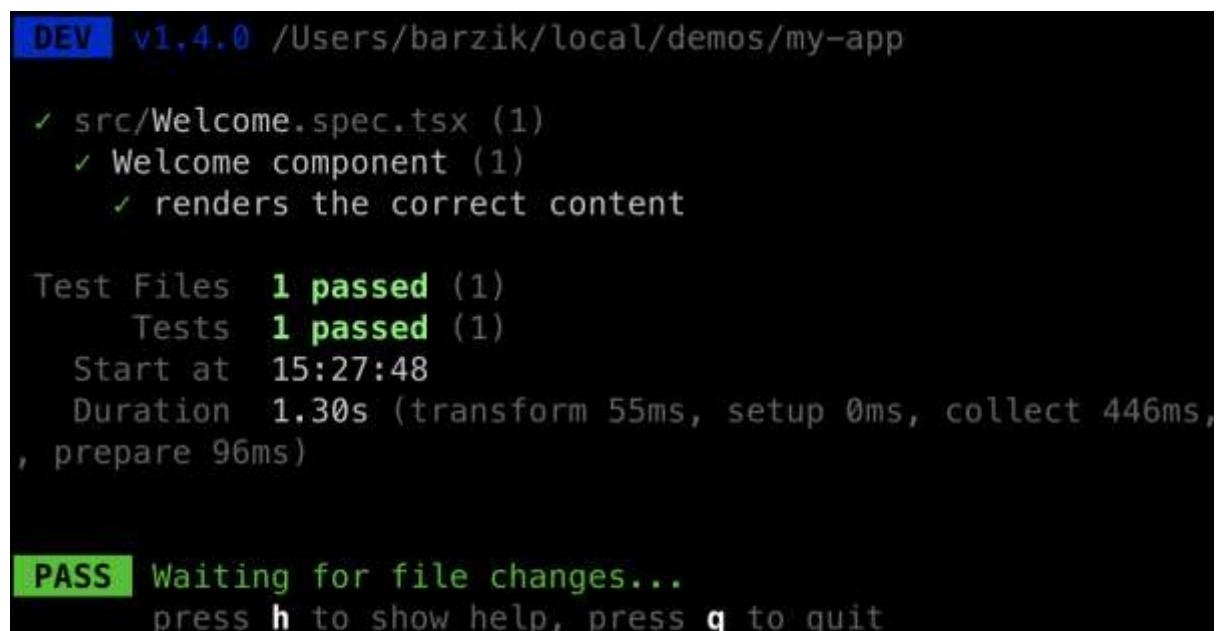
האמת היא שבמקרה זהה ההסבר הרבה יותר מסורבל מהקוד:

```
expect(screen.getByText('Hello world!')).toBeInTheDocument();
```

הקוד הזה מדבר בעד עצמו והוא מובן מאוד.

אחרי שכתבנו את הבדיקה הגע הזמן לראות שהיא עוברת בהצלחה. את הבדיקות אלו מרצים בטרמינל שכבר למדנו לעבוד איתו בפרקם קודמים. אם אלו עובדים עם Visual Studio Code או נפתח חלונית של טרמינל באמצעות בחירה בלשונית Terminal בחלק העליון של התוכנה ואז ב>New Terminal

לחופין, אתם יכולים לפתוח cmd ולנoot אל תיקייתuproject. לא משנה באיזו דרךם, על מנת להריץ את כל הבדיקות יש להקליד: `npx vitest`. מיד יירץ התהיליך ואם הבדיקה תעבור בהצלחה, תראו אישור.



```
DEV v1.4.0 /Users/barzik/local/demos/my-app

✓ src/Welcome.spec.tsx (1)
  ✓ Welcome component (1)
    ✓ renders the correct content

Test Files 1 passed (1)
  Tests 1 passed (1)
Start at 15:27:48
Duration 1.30s (transform 55ms, setup 0ms, collect 446ms,
, prepare 96ms)

PASS Waiting for file changes...
press h to show help, press q to quit
```

אם הבדיקה לא תעבור בהצלחה, תוכלו לראות את הסיבה שבגלה היא נכשלה:

```
Failed Tests 1

FAIL src/Welcome.spec.tsx > Welcome component > renders the correct content
TestingLibraryElementError: Unable to find an element with the text: Hello World!. This could be because the text is broken up by multiple elements. In this case, you can provide a function for your text matcher to make your matcher more flexible.

Ignored nodes: comments, script, style
<body>
  <div>
    <p>
      Hello world!
    </p>
  </div>
</body>
> Object.getElementError node_modules/@testing-library/dom/dist/config.js:37:19
> node_modules/@testing-library/dom/dist/query-helpers.js:76:38
> node_modules/@testing-library/dom/dist/query-helpers.js:52:17
> node_modules/@testing-library/dom/dist/query-helpers.js:95:19
> src/Welcome.spec.tsx:9:19
  7|   it('renders the correct content', () => {
  8|     render(<Welcome />);
  9|     expect(screen.getByText('Hello World!')).toBeInTheDocument();
  |
10|   });
11| });


```

כאן למשל טעיתי בכתיבת הבדיקה ובמקום לבדוק שהקובומפונטה מדפסה Hello world ב-w קטנה, הנחתי שהיא מדפסה Hello World ב-W גדולה. רואים את השוני.

## בדיקות לקומפוננטה עם props

אם לקומפוננטה יש props, علينا להעביר אותם בבדיקה. זה קל יותר ממה שזה נשמע. הנה נבחן את הקומפוננטה זו:

```
function Welcome(props: {name:string, surName:string}):JSX.Element {
    return (
        <p>Hello, {props.name} {props.surName}!</p>
    );
}

export default Welcome;
```

כאן נעשה שימוש ב-props. כיצד אבודק את הקומפוננטה? ראשית, אצור את קובץ הבדיקות. כמעט דבר לא משתנה מלבד הרנדור של הקומפוננטה. מנ הסתם אני ארצה לרנדר אותה עם ה-props המתאיםים. איך עושים את זה? כך:

```
import { describe, it, expect } from 'vitest';
import { render, screen } from '@testing-library/react';
import '@testing-library/jest-dom';
import Welcome from './Welcome';

describe('Welcome component', () => {
    it('renders the correct content', () => {
        render(<Welcome name="Moshe" surName="Cohen" />);
        expect(screen.getByText('Hello, Moshe
Cohen!')).toBeInTheDocument();
    });
});
```

השוני העיקרי הוא השורה זו:

```
render(<Welcome name="Moshe" surName="Cohen" />);
```

אם אני בוחן props, אני נדרש להעביר אותם ולבחון את התוצאה.

**not**

כשאנו מוסיפים את הפקציה `not` לפניה פונקציית ההשוואה, אנו בעצם הופכים אותה. הבדיקה הזאת, למשל, בודקת שהטקסט הוא לא `Error`:

```
expect(screen.getByText('Hello, Moshe  
Cohen!')).not.toBeInTheDocument();
```

**תרגיל:**

נתונה הקומפוננטה Greeting. כתבו לה בדיקה שבודקת שהוא מדפיס את הפלט Good MorningMoshe! כשם עביריים לה את ה-name Moshe!

```
function Greeting(props :{name: string}) :JSX.Element {
  return (
    <p>Good Morning {props.name}!</p>
  );
}

export default Greeting;
```

**פתרון:**

```
import { describe, it, expect } from "vitest";
import { render, screen } from "@testing-library/react";
import "@testing-library/jest-dom";
import Greeting from "./Greeting";

describe("Welcome component", () => {
  it("renders the correct content", () => {
    render(<Greeting name="Moshe" />);
    expect(screen.getByText("Good Morning
Moshe!")).toBeInTheDocument();
  });
});
```

בדוק כמה בדוגמה שהובאה בפרק.

פרק 23

# יצירת בילד והעלאת האפליה לסבינה חייה



# יצירת בילד והעלאת האפליקציה לסביבה חייה

אחרי שכתבנו את אפליקציית הריאקט שלנו – כתבנו קומפוננטות משלנו, השתמשנו בקומפוננטות חיצונית, כתבנו שירותים ופונקציות – הגיעו העת להעלות אותה אל השרת. צריך לזכור שבסוף כל יום כל האפליקציה היא קובץ ג'אווהסקריפט, קובץ HTML וקובץ CSS שנטענים על ידי הדף. בקובץ הג'אווהסקריפט זהה, הדחוס והמחובר, יש הנול: ספריות ריאקט, babel, הקומפוננטות החיצונית והקוד שנכתבם. הכל בקובץ אחד. הדף טוען אותו והכל מתחילה היבנות ממש. יש-Calculators שבודדים חלקים – למשל תלויות חיצונית (ספריות צד שלישי כמו למשל UI React Material שעלייה למדנו באחד הפרקים הקודמים) והאפליקציה שכתבנו. אבל בסופו של הבילד, התוצאה היא קבצים דחוסים.

הקובץ הזה אמר להגיע לדף אינטרנט. כשאנו בסביבת הפיתוח, עם Vite, האפליקציה יוצרת לנו שרת קטן על המחשב (mbased.js), יוצרת לו כתובת (localhost:5173) ואיפילו פותחת את הדף ומנוה לאתר הזה שטוען בתורו את קובץ הג'אווהסקריפט. אבל מה קורה באתר אמיתי?

באטר אמיתי אנו צריכים שרת אמיתי שיגיש את הקבצים לדף. כשהאני אומר "יגיש" אני מתכוון לתהילך שבמסגרתו אנו מקלידים כתובת של אתר אינטרנט בדף. הדף ניגש לשרת האתר והוא מחזיר לו קבצים שהדף מציג.

כך למשל כשהאני גולש לגוגל, אני מקליד בדף google.com, השרת של גוגל מעביר לדף דף HTML, קובץ ג'אווהסקריפט ואת הלוגו של גוגל (או כל דודל אחר של גוגל שיש שם במקום הלוגו). בסופו של דבר יש שרת שמעביר לדף קבצים.  
אילו קבצים?

1. דף HTML שיש בו קישור אל:
2. קובץ CSS של עיצוב.
3. קובצי הג'אווהסקריפט המכוזרים של האפליקציה.

איך אנו יוצרים את השרת זהה? יש הרבה דרכים. הדרך פשוטה ביותר הוא לשכור shared hosting באתר כלשהו ופשוט להציב עליו את קובץ ה-.html index וקובצי ה-CSS והג'אוסקריפט (זה עובד אם עושים רואוטינג מבוסס #), באמצעות העתקה פשוטה, בתיקייה הראשית של השרת בעזרת הממשק של ניהול השרת שספק השירות מעניק לכם. הקולדת כתובה שם המתחם וכניסה אל האתר תגרום לטעינת קובץ ה-HTML ומכובן כל הקבצים המוקשרים אליו.

יש דרכים רבות אחרות – לא חסרות שפوتצד שרת שיש בהן שירותים. הפופולרית ביותר היא Node.js, אך יש גם את ג'אווה, #C ופייתון שמאפשרות יצירת שירותים. אם אתם, או מתכנתים לצד השרת בחברה שלכם, יוצרים שרת כזה – בסופו של דבר תצטרכו לגרום לו להחזיר ללקוח את דף ה-HTML והקבצים. הלקוח ייכנס, הקבצים ייטענו ומפה התפקיד האפליקציה לפעילה בדיקון כמו vite.

איך מייצרים את קובץ ה-HTML, קובץ CSS ומכובן קובצי הג'אוסקריפט? באמצעות פקודת wkhtml מינימל – ממשק הפוקודה הטקסטואלי. אם אתם משתמשים ב- New Terminal, Visual Studio Code, לחזו על Terminal בתפריט העליון ואז על Code. יפתח לכם חלון בתחום הממשק עם שורת פוקודה טקסטואלית.

אם אין לכם Visual Studio Code, היכנסו אל ה-prompt בחלונות ונווטו אל התיקייה של האפליקציה.

מיד כשאתם יכולים להקליד בטרמינל, הקלידו:

```
npm run build
```

ירוץ תהליך שמאחד ומכווץ את קובצי ה-`index.html` ו-`CSS` ויצרת קובץ `index.html` וגם קבצים סטטיים נוספים שמשתמשים לשרת. כל הקבצים האלה ייכנסו אל תיקיית `dist` – זו התקינה שיש להעלות אל השרת כדי להשתמש באפליקציה שלכם, בין שמדובר בשרת של חברה גדולה ובין שמדובר ב-`shared hosting`:

```
> my-app@0.0.0 build
> tsc && vite build

vite v5.2.8 building for production...
✓ 39 modules transformed.

dist/index.html          0.46 kB | gzip: 0.30 kB
dist/assets/index-DiwrgTda.css 1.39 kB | gzip: 0.72 kB
dist/assets/index-Em8Uhd_8.js 164.15 kB | gzip: 53.45 kB
✓ built in 767ms
```

התהליך הזה נקרא בילד והוא נפוץ באפליקציות צד לקוח כמו ריאקט, אングולר ודומותיהן. התהליך הזה מתרחש אוטומטית כאשר מתכנת מוסיף קוד לפרויקט וההתהליך משולב בדרך כלל עם הבדיקות האוטומטיות.

אפשר לבדוק את הבילד באמצעות מודול `npm run serve`. מתקינים אותו גלובליות באמצעות הקלדת הפקודה זו בטרמינל:

```
npx serve -s dist
```

וכניסה אל:

localhost:5173

ההבדל בין השיטה זו להקלדת dev run הוא שכאן אנו מרים את הבילד ממש, כלומר את הקבצים המוגמרים שאנו אמורים להעלות לשרת. מודול serve יוצר לנו שרת ומעלה את התקינה dist כדי שנוכל לבדוק את האפליקציה שלנו, מהיקות ה-dist, מקומית.

**שימוש לב:** אם כאשר אתם מקלידים serve אתם מקבלים הודעה שגיאה בנושא:

serve : The term 'serve' is not recognized as the name of a cmdlet

**עשו את הפעולות הבאות:**

הקלידן:

npm install serve

(לא ה-g)

ולאחר מכן הקלידן:

node .\node\_modules\serve\bin\serve.js -s build

**תרגיל:**

קחו כמה קומפוננטות שיצרנו בתרגולים הקודמים ושלבו אותן באפליקציית הריאקט שיש ב-`App.js`. הריצו את הבילד ובדקו אותו.

**פתרון:**

יש להריץ `npm run build` ולהכנס לתוצאה. אחרי שהתהליך מושלם, תיווצר אצלכם תיקיית `dist` עם קובץ `index.html`, קובצי תמונה, `CSS` והכى חשוב: ג'אווהסקרייפט.

על מנת לבדוק את הבילד, אפשר להשתמש ב-`serve` מקומי כפי שלמדנו. תיפתח לכם בדף אפליקציה מלאה שנמצאת ב-`localhost`. אפשר להעלות אותה לכל שרת באמצעות העתקת תיקיית `dist` והעברתה לשרת מבוסס `Node.js`, `Apache` או כל שרת אחר.

פרק 25

# סיום - ומה ענשין?



## סיום – ומה עכשווי?

אם קראתם את כל פרקי הספר, אתם יודעים לבנות אפליקציות ריאקטיות. אבל הידע שלכם הוא עדין תיאורטי, ולימוד פרימומורק חדש נעשה באמצעות הידיים. אי-אפשר ללמד פרימומורק אם לא עברתם את השלבים שאף ספר לא יכול להבהיר אתכם: התסכול המתלווה למשהו שלא עובד כשורה, השמחה הגדולה כאשר מצליחים לפטור את התקלה ורוכשים ידע נוסף יקר מפז, החיפוש המתמיד בגוגל, ברכי בינה מלאכותית ו-[stackoverflow](#) ובנויות מוצרים אמיתיים ופתרון התקלות שיש בהם.

אם סיימתם לקרוא את הספר ולפטור את התרגילים, הצעד הבא שלכם הוא לבנות אתר אפליקציה או יישום מבוססי ריאקט. זה הזמן להצטרף לקבוצה שבונה אתרים או אפליקציות בהתקנות עבור מיזמים הקרובים ליביכם, לבנות אתר או אפליקציה לחבר שתמיד יהיה לו רעיון או לעמotaה שתמיד רציתם לתרום לה. הניסיון המעשוי הראשון שליכם לבניית אתרים ובכתיות קוד היה עבור עמותת טולקין (מחבר "שר הטבעות") בישראל – אין סיבה שהה לא יהיה כך עבורכם. הניסיון כאן הוא קריטי ואתם חייבים לבנות משהו משל עצמכם, אפילו ברמת הניסיון. אסור להסתפק בתרגילים שיש בספר.

כאשר אתם מחליטים לבנות משהו משלכם – מילת המפתח היא תכנון. אחד מנהלי הפיתוח שלי נהג לומר ששבועיים של תכנונות חוסכים שעתיים של תכנון. חשבו על הקומפוננטות, בדקו באילו קומפוננטות חיצוניות אפשר להשתמש ואיך הן עובדות ואם אפשר להשתמש בהן עבור האתר שלכם. בנוסף לכך, בקשו לקבל ביקורת על העבודה שלכם וכך תגלו איך תוכלו יכולות לעשות אותה טוב יותר.

אבל אחרי שבניתם את הפרויקט שלכם – יש עוד כמה צעדים שתצטרכו לעשות כדי להשתלב בשוק העבודה. אף אחד לא יוכל אתכם לעובדה על סמך קריית הספר הזה ופרויקט גמר, מוצלח ככל שהיא. יש כמה נתיבים נוספים שיאפשרו לכם להמשיך ללמידה ולצבור ניסיון – עוד לפני העבודה הראשונה. **בספר "פיתוח ווב מעשי בעברית" אני מסביר על כיצד יוצרים ניסיון כזה ומשלבים אותו בקורות החיים.**

## המחברות להילת הפיתוח

אחרי שבנויתם את הפרויקט שלכם, או במהלך הפיתוח שלו – חשוב גם להתחבר להילת הפיתוח הישראלית. היא חמה ונעימה ויש לא מעט מתקנים שיעזרו לסייע לכם המשיך הלאה. בפייסבוק יש קבוצה פULAה מאוד של מתקני ריאקט שפועלת בעברית וכן לא הטרף אליה ולהשתתף (בענוה הרואה) בדיונים, לקרוא, ללמוד ולהבין לאן כדאי להتقدم, ואם יש פרויקט פתוח שמחכה למתרדים – נסו להטרף אליו.

## פגשים ומיטאפים

באטר [meetup.com](http://meetup.com) יש לא מעט מידע על מפגשים בחינם שנקראים מיטאפים. במפגשים האלו נפגשים מתקנים ושותפים הרצאות או בונים יחד דברים. יש מפגשים המיעדים למתקני ריאקט וכן לヒרים לקבוצות השונות שמארגנות מפגשים על מנת לדעת על אלו שמתקיימים בסביבתכם ובשעות המתאימות לכם. במפגשים האלו אפשר גם להמשיך ללמידה על פיצרים שונים בריאקט וגם להיפגש עם מתקנים מקטועים, לשאול לעצם ולקבל הכוונה בנוגע להתקדמות בלימוד השפה ובפרויקטים שונים. מדי פעם גם יש באתר הזמנה להاكتונים ולמפגשים שבהם מתקנים בפועל. כדאי מאוד להצטרף לקבוצה צו ולתיכנת לצד מתקנים אמיתיים בליוי הכוונה אמיתית. כמה שעות עבודה ככלו יקומו אתכם באופן מטורי, גם אם לא ת贔ו בפרס.

## האתר Stackoverflow

האתר [Stackoverflow](http://Stackoverflow.com) הוא שימושי בשתי דרכים – ראשית הוא מכיל תשובות לשאלות רבות על ריאקט וסביר להניח שתיתקלו בו כאשר תחפשו בגוגל שאלות שונות על ריאקט. אבל הוא חשוב גם כי הוא מאפשר לכם לענות על שאלות של מתקנים אחרים, שאולי פחות מנוסים מכם בריект בתחום התיאורטי. השתתפות בדיון וניסיון לענות על שאלות של אחרים בריект גם יקדםו מאוד את הידע שלכם. פרופיל עשיר ב-WStackoverflow הוא משאנו שראוי לציין בקורות החיים.

## תרומת קוד בגייטהאב

לבסוף, אפשר לתרום קוד בגייטהאב לפרויקטים שימושיים בריאקט. למשל, ספריית קומפוננטות שmbosst על ריאקט. בגייטהאב יש לא מעט פרויקטים שימושיים לתרומת קוד. במהלך הפרויקט שבנותם בוודאי ראתם ספריות קומפוננטות שחרסה להן בדיקת הקומפוננטה הזו שהייתה רציתם. למה שלא תוסיפו אותה? או תתקנו את הדוקומנטציה? או תכתבו בדיקה אוטומטית במקומם שבו היא חסירה – תרומת הקוד הזו היא ממש בבחינת ניסיון, ואם תבחרו בנתיב הזה ותצליחו בו, תוכלו אפילו להשתלב בקלות בתעשיית התוכנה.

מצד שני, העבודה בגייטהאב מצריכה ידע בגייט, הבנה באנגלית ויכולת לתקשר עם המתכנתים שעובדים בפרויקט ומגיעים מכל העולם. יש גם מיטאפים שבהם מתכנתים אחרים מסבירים איך עובדים עם גייטהאב ותרומים קוד. בספר שלי "לימוד פיתוח מעשי בעברית" אני מלמד גם על גיט ואיך תורמים קוד לגייטהאב על מנת לבנות TICK עבודה מסודרת.

כך או כך – הספר הזה הואצעד הראשון בכונסה לעולם המרתך של ריאקט ופיתוח צד ללקוח. אני מאמין לכם הצלחה בלימוד וגם בהשתלבות בתחום.

# נספח: **PropTypes**

מחבר: רן בר-זיק עבור חברת HoneyBook

התקשרות בין קומפוננטות שונות נעשות באמצעות `props` שמעבירים קלט או, באמצעות פונקציות, פלט. על מנת להשתמש בקומפוננטה, אנו צריכים לדעת באיזה `props` להשתמש. מה קורה אם מי שימוש (`או, במקרה מסוים יותר, צורך`) את הקומפוננטה שלנו לא משתמש-ב-`props` המתאים? הקומפוננטה לא תעבוד או שתעבד באופן לא צפוי. אם אנו כותבים קומפוננטה, החובה שלנו היא לסמן לאלו שימושים בה (גם אם המשתמשים בה הם אנחנו, בעוד) באיזה `props` אפשר להשתמש. כדי שמי שימוש בקומפוננטה יקבל חיוו מיידי. איך עושים זאת? באמצעות `PropTypes`. מודול שהוא חלק מריאקט עד גרסה 15.5 והחל מהגרסתה זו הוא יצא למודול משלהו שנitinן להתקינו בנפרד. `PropTypes` הוא הדרך שלנו להציגו איזה `props` אנו נדרש לקבל ומהו סוג. כך למשל, אני יכול להציג על `props` מסוימים שדרישים ועל כלו שהם רק אופציונליים וכמובן לפרט את הסוגים שלהם. כך, אם מי שימוש בקומפוננטה שונח להעביר `props` מסוימים, הוא יקבל התראה. אם הוא מעביר מחרוזת טקסט ועוד אנו מצפים למספר, הוא מקבל גם כן התראה מאוד ברורה.

למרות ש-`PropTypes` אינם חלק מריאקט, רבים מהתוכננים שפתחים בריект עובדים איתם ומומלץ מאוד לעשות כן.

## התקנת `PropTypes`

בדוק כמה ספירת קומפוננטות חיונית, שלמדנו עלייה בעבר, או `Create-React-App`, `npm`. אנו מתקינים את `PropTypes` באמצעות `Node.js` והכלי שלו להתקנת מודולים: `npm`. בהנחה ש-`Node.js` מותקנת במחשבם ואותם משתמשים באפליקציה `Create-React-App`, הכנסו באמצעות הטרמינל לתיקיה הראשית של האפליקציה שלכם והקלידו את הפקודה הבאה:

```
npm install prop-types
```

אם הכל תקין, לאחר נדקה תוכלו לראות בטרמינל חיוי על כך שההתקנה האличה. מהנקודה הזו תוכלו להשתמש ב-`.PropTypes`.

```
* my-app git:(master) ✘ npm install prop-types
WARN @typescript-eslint/eslint-plugin@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
WARN @typescript-eslint/parser@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
WARN ts-pnp@1.1.2 requires a peer of typescript@* but none is installed. You must install peer dependencies yourself.
WARN tsutils@3.17.1 requires a peer of typescript@>2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.5.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.
+ prop-types@15.7.2
updated 1 package and audited 983897 packages in 7.798s
```

## השימוש ב-`PropTypes`

השימוש נעשה בקוד שבו אנו כותבים את הקומפוננטה. בין אם מדובר בקומפוננטה מבוססת קלאס או קומפוננטה מבוססת פונקציה. אנו מיבאים את `PropTypes` באמצעות `import`, בדיקות כמו כל ספירה חיצונית אחרת של מדנו עליה ובאמצעות האובייקט שבו מיבאים אנו מצהירים על ה-`props` השונים.

הבה ונדגים באמצעות הקומפוננטה הפешוטה זו:

```
function Welcome(props: {name:string}):JSX.Element {
  return (
    <p>Hello, {props.name} !</p>
  );
}
export default Welcome;
```

אם אנו רוצים להשתמש בקומפוננטה זו, אנו נעשה זאת, למשל, כך:

```
import './App.css';
import Welcome from './Welcome';

function App(): JSX.Element {
  const userName:string = 'Moshe';
  return (
    <div className="App">
      <header className="App-header">
        <Welcome name={userName} />
      </header>
    </div>
  );
}

export default App;
```

אבל מה קורה אם אנו לא מעבירים את `props` המתאים? למשל אם אנו לא מעביר `name`?  
זו תקלה שיכולה להתקיים. בשלב שבו כתבתי את קומפוננטת `Welcome` עד השלב שאני משתמש בה יכולם לעبور החדשמים, אולי שניים. אם אשכח את מה שאני אמרו להעביר - מה שיקרה הוא שהקומפוננטה לא תעבור כשרה ולי לא יהיה מושג למה. אני אצטרך לנבור בקוד כדי להבין מה השتبש.

הבה ונshall מושב ב-`PropTypes`. ראשית ייבוא ואז הצהרה:

```
import PropTypes from 'prop-types';

function Welcome(props:{name: string}):JSX.Element {

  return (
    <p>Hello, {props.name} !</p>
  );
}

Welcome.propTypes = {
  name: PropTypes.string.isRequired,
}

export default Welcome;
```

היבוא נראה כך:

```
import PropTypes from 'prop-types';
```

הזהרה נראה כך:

```
Welcome.propTypes = {
  name: PropTypes.string.isRequired,
}
```

הזהרה היא בעצם אובייקט בשם `propTypes` שבו אני מצמיד לקומפוננטה שלי. האובייקט מכיל את כל ה-`props` ואת הסוג שלהם. הסוג מורכב משני חלקים לפחות:

`PropTypes.string.isRequired`,

החלק הראשון, **PropTypes** הוא מנדטורי. החלק השני הוא סוג **the prop**. חלק שלישי הוא האם מדובר ב-**prop** שהוא נדרש לפעולה תקינה של הקומפוננטה.

אם אני אנסה להשתמש בקומפוננטה בלי להציג ערך `name`. אני מקבל הודעה שגיאה מאוד ברורה שמודיעה לי על כך ש-`prop name` מסומן כנדרש אך הערך שלו הוא לא מוגדר.

```
✖ Warning: Failed prop type: The prop 'name' is marked as required in 'Welcome', but its value is 'undefined'.
  in Welcome (at App.js:10)
  in App (at src/index.js:16)
  in Provider (at src/index.js:16)
```

אם אני אעביר סוג נתון שאינו תואם לסוג שאני מבקש, אני מקבל גם הודעה שגיאה ברורה שמודיעה לי ש-`prop name` קיבל ערך שאינו תואם את מה שהקומפוננטה רוצה לקבל.

```
✖ Warning: Failed prop type: Invalid prop 'name' of type 'number' supplied to 'Welcome', expected 'string'.
  in Welcome (at App.js:10)
  in App (at src/index.js:16)
  in Provider (at src/index.js:16)
```

למרות הודעה השגיאה – הקומפוננטה תמשיך לרווח כרגיל ותנסהランדר את הפלט. אם לא יהיו שגיאות נוספות – השגיאות ש-**PropTypes** מציגה לא יגרמו לקומפוננטה לא לרווח.

## ערכים שאפשר לקבוע

אנו יכולים להודיע באמצעות `PropTypes` על מגוון רב של סוגי נתונים שונים – כמעט כל הנתונים הprimיטיביים האפשריים. בטבלה זו יש ריכוז של סוגי הנפוצים:

סוג הערך	תיאור	דוגמה
מחרוזת טקסט פשוט	מחרוזת טקסט פשוטה	<code>PropTypes.string</code>
מספר	כל מספר שהוא	<code>PropTypes.number</code>
בוליאני	<code>false</code> או <code>true</code>	<code>PropTypes.bool</code>
פונקציה	כל פונקציה, רגילה או אונומית	<code>PropTypes.func</code>
מערך	מערכות המכילים מידע, כולל מערך ריק	<code>PropTypes.array</code>
אובייקט	אובייקטים שמכילים מידע או מתודות, כולל אובייקטים ריקים	<code>PropTypes.object</code>
אלמנט ריאקט	קומפוננטה ריאקטית כלשהי שימושה כפרמטר	<code>PropTypes.element</code>

אני יכול להחליט שה-`Type` שלי יקבל כמה סוגיים. כך למשל, אם הקומפוננטה שלי יודעת להתחזק עם מספר או טקסט ב-`prop` מסוים. אני יכול לציין שאני יכול או סוג אחד, או סוג שני. אני עושה את זה באמצעות הפקודה `oneOfType` במקומ הסוג.

למשל:

```
import PropTypes from 'prop-types';

function Welcome(props: {name: string}): JSX.Element{

  return (
    <p>Hello, {props.name} !</p>
  );
}

Welcome.propTypes = {
  name: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number,
  ]).isRequired,
}

export default Welcome;
```

אני תמיד יוכל לשרש את `isRequired` במידה והפרמטר נדרש. אך זו לא חובה.

חשוב להזכיר `PropTypes`. מדובר בתוספת קטנה וחשובה מאוד לכל קומפוננטה שיכולה לעשות סדר בכך. נכון, זה מעט יותר עבודה, אבל לטווח ארוך זה חשוב וכך.

## **נספח: שינויים מהמהדורה הקודמת (הדורה 2.0)**

כיוון ש Create React App הוצאה למילואות, החלפתו את כל הדוגמאות של האפליקציות ל-Vite – כולל Jest שהועברה ל-Vitest. תיקוני שגיאות בקוד של דוגמאות שונות: פרק 10, 11, 21 – תודה רבה לנתנאל כהן שريقץ את כל השגיאות. הוסיף text alternatives לתמונות וצלומי המסך על מנת לבצע הנגשה יותר טובה.

## **נספח: שינויים מהמהדורה הקודמת (הדורה 1.1.1)**

בגרסה הוכנסו שינויים משמעותיים ללמידים ריאקט עם טיופסקרייפט ולא עם ג'אווהסקייפט. כל הדוגמאות הוחלפו. הוכנסו שדרוגים והתאמאה לריאקט 18 – החלפת מתודות ישנות, הסבר על כך שעדייף שלא להשתמש בקלאסים. הוכנסו שדרוגים המתאימים לגרסה האחורונה של react-router ו-*ui material* (מאי 2023).

**הוסר הפרק על רידקס** כיון שהפופולריות של רידקס כבר לא בשיאה, הוא מורכב מדי ללמידים חדשים וקונטיקסט כבר פופולרי בהרבה. באופן אישי למדתי לטעב את רידקס בלהט, אני כבר לא יכול ללמוד אותה.

## **נספח: שינויים בין המהדורה 1.1.1 لامהדורה 1.0.0**

פרק הרידקס הוחלף לחלוטין בדוגמה פשוטה יותר. בדוגמה החדשה אין אובייקט `list` שבתוינו מערך של `items` אלא מערך של `items` בלבד. בנוסף, הוננה אפליקציית ריאקט ב `sox.io` ו קישור אליה הוצב בסוף הפרק.

נוספַּת תרגיל לפרק הרידקס.

הדוגמה בפרק על הסטייט הוחלפה לדוגמה הכלולת אירוע ולחיצה במקום דוגמת `setInterval`.

תיקוני שגיאות הקלדה בפרק אודות המרצים ואודות Honeybook.

עדכון הדוגמאות והקישורים בפרק הפתיחה לריאקט גרסה 17.

הורדת הדוגמאות בשל השדרוג לגרסה 17.

הוספה משפט ודוגמה ב-`React.Fragment` על הקיצור המקביל: `<>`.

תודה רבה על הרכישה של הספר!

עבדתי מאוד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוכרי העריכאה. יותר מ-1800 אנשים תמכו בספר זהה ואיפשרו לו לצאת לאור.

הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהקינדל, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתוך תקווה שהרוכש והותמן לא ינצל את האמון שנתתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפיצה שלו. אני מאמין שרוב האנשים הוגנים.

העתק זהה נמכר ל:

nivbuskila@icloud.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נჩבים-items פרטי הרוכש באופן שוקוף למשתמש. כדאי מאוד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיהקנו את העתק שנותץ ברשותכם.

תודה וקריאה נעימה!