

תודה רבה על הרכישה של הספר!

עבדתי מאוד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוצריו העריכה. יותר מ-1800 אנשים תמכו בספר זהה ואייפשרו 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. מתווך תקווה שהרוכש וה透מך לא ינצל את האמון שנתי בudentה העותקה סיטונאית של הספר לאנשים אחרים והפצה שלו. אני מאמין שרוב האנשים הוגנים.

הઉתק הזה נמכר ל:

nivbarsh@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נחברים פרט' הרוכש באופן שקוֹף לשימוש. כדי מאד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למי שהוא אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיחקנו את העותק 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: 1.1.0



הקויה האקדמית אונ

Ono Academic College

החוג לחדש



Really Good



HoneyBook

כל הזכויות שמורות © רן בר-זיק, 2021.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רישיון לא-בלודי, לא-יחיד, איש, בלתי ניתן להעברה (למעט על פי דין), ובلتוי ניתן להסבה לעשות שימוש אישי בספר זה לצרכים לימודיים בלבד.

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

מותר לך לצטטקטעים קצרים מהספר במסגרת השימוש הוגן, כמו פרוסה 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 בתוך תוכנות שתפתח. אם אתה רוצה להכנס אל פוריקט שלו, שלח מייל ונדבר על זה.

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

הפקה: כריכה – סוכנות לסופרים
www.kricha.co.il



תוכן העניינים

10	על "לימוד ריאקט בעברית"
11	על המחבר
12	על העורכים הטכניים
12	דורון זבלבסקי
12	gil pinck
13	דורון קילוי
14	על החברות התומכות
14	Really Good
14	אלמנטור
15	HoneyBook
16	על ריאקט
18	דרכן הלימוד
18	על המונחים בעברית
19	סביבת העבודה הבסיסית
22	סביבת עבודה בסיסית-ב-codepen
25	אפליקציית ריאקט
31	בנייה קומפוננטת פונקצייה בסביבת העבודה הבסיסית
40	בנייה של סביבת ריאקט אמייתית ומורכבת יותר באמצעות Create React App
40	התקנת Node.js על המחשב שלכם
40	התקנה על חלונות
44	התקנה על מק
44	התקנה על לינוקס
45	עבודה עם טרמינל
45	הפעלת הטרמינל
46	גיוסט בטרמינל
48	מציאת מיקומים בטרמינל דרך חלונות
49	טרמינל ב-Visual Studio Code
49	בדיקות גרסאות Node.js דרך הטרמינל
50	עבודה עם Create React App
54	קשיים ותקלות

57	כתיבת קומפוננטה ראשונה ב-React App Create
66	Export \ Import
69	יבוא מנתיבים אחרים
71	אין חובה להשתמש בסימט הקובץ js
71	"יצוא של כמה משתנים
72	"יבוא של תמונות, CSS ומשאים אחרים
75	JSX
81	רשימות ב-JSX
91	קומפוננטה עם תכונות (<i>props</i>)
105	דיבאג
110	סמיין
112	ריאקט ורנדור
113	סטייט
121	תכנון מבנה הקומפוננטות
126	איروسים ועדכון קומפוננטות
127	אירוע DOM
128	איروسים סינטטיים / ריאקטיים
149	אלמנט <i>fragment</i>
152	useEffects
158	קומפוננטה ללא שם
161	עיצוב קומפוננטות
163	CSS בסיסי
163	סלקטו
166	תובנות
170	קלאס ריאקטוי
180	מעגל החיים בקלאס
180	componentDidMount
180	componentDidUpdate (prevProps, prevState, snapshot)
180	componentWillUnmount
180	shouldComponentUpdate (nextProps, nextState)
181	getDerivedStateFromError (error)
181	componentDidCatch (error, info)

189.....	שימוש בקומפוננטות מקורות אחרים.....
194.....	יבוא כמו קומפוננטות.....
196.....	מודולים חסריים.....
209.....	HOC: Higher Order Component.....
223.....	ראונינג.....
246.....	קונטנסט.....
259.....	חיבור לשרת עם סרוויסים.....
267.....	מבוא לבדיקות עם jest.....
269.....	חלק ראשון – import.....
269.....	חלק שני – בתייה מסגרת הבדיקה.....
269.....	חלק שלישי – הרצת הקומפוננטה.....
269.....	חלק רביעי – הבדיקה.....
272.....	בדיקות לקומפוננטה עם props.....
273.....	פונקציית בדיקה – assert.....
273.....	not.....
273.....	toBeTruthy.....
273.....	toBeFalsy.....
273.....	toContain.....
278.....	יצירת בילד והעלאת האפליקציה לserveה חייה.....
284.....	רידקס.....
285.....	הפילוסופיה מאחורי RIDKS.....
285.....	קומפוננטות מנהלות מול מנהלות.....
292.....	הטמעת RIDKS.....
292.....	התקנת RIDKS.....
292.....	תכנון הסטייט וזרימת המידע במערכת.....
294.....	mapStateToProps.....
295.....	mapDispatchToProps.....
295.....	הימוש של RIDKS במערכת שלנו.....
311.....	סיכום – ומה עכשוו?.....
311.....	התחברות לקהילת הפיתוח.....
311.....	מפגשים ומיטאפים.....
312.....	האתר Stackoverflow.....

312.....	תרומת קוד בGITהאב
313.....	נספח: PropType
313.....	התקנת PropTypes
314.....	השימוש ב-PropTypes
318.....	ערכיהם שאפשר לקבוע.....
320.....	נספח: שינויים מהמהדורה הקודמת (1.0.0) (מהדורה 1.0.0)

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 שלה, כולל קומפוננטות רבות המלויות אותה, כל מתכנת יכול ליצור אתר או אפליקציה מורכבים בmphירות רבה ובאיכות יוצאת דופן. לריאקט יש גם מעתפת (כמו אלקטрон או קוורדובה) המאפשרת למשק שפותח לוויב להיות מותאם בקלות גם לתוכנות מחשב של ממש. העושר של האקויסיטם הפיתוחי של ריאקט וחוזק הקהילה שלה בארץ ובעולם מאפשרים לקבל גם המון-המון מידע ווועב בכל תקלה ובעיה. אם מתכנת נתקל בעיה בפיתוח, סביר להניח שהוא ימצא סיוע ומידע בפורומים ובקבוצות מתכנתים. יש גם שפע של מיטאים וקבוצות דיוון המוקדשים למפתחים העובדים בריאקט ונוסף על כך, הביקוש למפתחים המכירים את ריאקט הוא גבוה. כל הדברים הללו הופכים את ריאקט לאידיאלית לכינסה לעולם פיתוח צד הלוקה.

הספר מניח ידע מקיף בג'אווהסקרייפט וידע בסיסי ב-HTML וב-CSS. אם איןכם מכירים ג'אווהסקרייפט, אפשר ללמוד זאת בספר "לימוד ג'אווהסקרייפט בעברית". יש שם פרק המסביר על HTML בסיסי ועל CSS בסיסי. מי שסימן לקרוא את הספר ההוא ותרגל כהלה, אמור להחזיק במידע מספק על מנת להתחיל ללמד ריאקט. בספר זה נלמד על ריאקט ממש מהבסיס – הכרת מונחים בסיסיים ויצירת סביבת עבודה – וגעע עד חומרים מתקדמים כמו `high order components` ו-`state hooks`. הספר מעודכן לגרסה ריאקט 17.

על המחבר

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

נסוף על עבודתו כמפתח במשרה מלאה, REN הוא עיתונאי ב"הארץ" 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 וסבירים על תכנות בעברית ומתעדכן לפחות פעם בשבוע.

REN הוא מחבר הספר "לימוד ג'אווה סקרייפט בעברית" והספר "לימוד JavaScript בעברית" ומלמד בקורס האקדמית אוננו.

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

על העורכים הטכניים

דורון זבלבסקי

הקריירה של דורון התחילה דוקא בצד השרת, בחברות אבטחת מידע שבהן עבד כמהנדס תוכנה, מוביל טכני ומנהל מוצר.

את המעבר לעולם הפורנטאנד ביצע בשנת 2014 כאשר הצטרף ל-Appitools, 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 וimplements אותה בחומר.

דורון הקים את קהילת ריאקט בישראל, כולל קבוצת פיסבוק ומיטהפ פופולרי, אף יוזם אתכנס ReactNext הראשון בישראל והוא מפיק משותף שלו.

נוסף על כך הוא מרצה על ריאקט, על בידיקות אוטומטיות ועל בניית צוותים מנכחים ותורם מזמננו בשמה למתחרים המבקשים סיוע בתחום במסגרת מפגשים אישיים וקבוצתיים.

gil Fink

gil Fink הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert והמייסד של חברת sparXys Microsoft Developer Technologies MVP.

כיום הוא מייעץ לחברות ולארגוני שונים, שם הוא מסייע בפיתוח פתרונות מבוססי אינטרנט -As SPAs.

הוא עורך הרצאות וסדנאות ליחידים ולחברות המעוניינים להתמחות בתשתיות, בארכיטקטורה ובפיתוח של מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט (Microsoft Official

"Pro Single Page Application Development" (Course MOC), מחבר משותף של הספר (Apress) AngularUP (Apress) ושותף בארגון הכנס הבינלאומי.

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

דורון קילזי

דורון ח' נושא פיתוח לרשת זה כעשור. את דרכו החל ביחידה טכנולוגית מובחרת בחיל המודיעין, שבה שירות 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 עוסק, בין היתר, במעבר של החברה לטכנולוגיות ותיקות כגון *Angular.js* ו-*tech-ad.js*. הדף של דורון ללימוד ולהשתפר גורם לו להמשיך להתקצע ולהתאהב בכל יום חדש בעולם.

על החברות התומכות

Really Good

Really Good היא בוטיק פיתוח Front End שעבדת עם סטארטאפים וחברות טכנולוגיות מאז הקמתה ב-2012 על ידי שחר טל ורוני אורבר. אנחנו נהנים לבנות אפליקציות מורכבות עם UX מוקפד במגוון טכנולוגיות ללקוחות מעניינים שחוויות המשתמש חשובות להם, ושומרים על איזון בריא בין עבודה לחיים.

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

ReallyGood.co.il

אלמנטור

אלמנטור מפתחת פלטפורמת קוד פתוח לבניית אתרים שמשנה את הדרך בה בנייתו אתרי אינטרנט בשוק המktezu. אלמנטור מעניק למשתמשים את החופש ליצור עמודי אינטרנט ללא צורך בקוד ולפתחים את היכולות לדוחף את הגבולות, לרענן ולהרחיב את המערכת בצורה קלה ומהירה באמצעות API ידידותי למפתחים, ובכך לחסוך זמן פיתוח ולהיות יעילים ורוווחיים.

עם מיליון+ אתרים הפעילים על אלמנטור וצמיחה חרודשית מדהימה, התגבשה סבב הפלטפורמה קהילתית חזקה המונה מאות אלפי חברות, מפתחים, משקוקים ומשתמשים, המקיימים מיטאים בכל רחבי העולם. מידי יום האלמנטוריסטים מייצרים וצורכים אלפי שעות של הדרכות, סרטוי השראה ובלוגים מעמיקים, ומפתחים תורמים קוד ורעיוןות באמצעות GitHub. האקויסיטם המktezu של אלמנטור מתפתח ללא הפסקה והוא אוצר המושך ומעשייר את היכולות של כל יוצר אינטרנט.

באלמנטור אנחנו משתמשים בטכנולוגיות קוד פתוח מתקדמות לפיתוח כל אינטרנט חדשניים ורוווחיים. אם גם אתם רוצים להיות חלק מהטכנולוגיה שמשנה את חווית האינטרנט בעולם ויש לכם את הידע כדי לבנות עולם יפה יותר אנחנו מתחשים אתכם, מעצבי UI&UX, מפתחי Full Stack, מהנדסי DevOps ו Big Data Kubernetes על פלטפורמות הענן של GCP & AWS.

אתר החברה: <https://elementor.com>

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

HoneyBook

חברת HoneyBook מפתחת פלטפורמה לניהול פיננסי ועסקן עבור עצמאיים ועסקים קטנים. החברה מאפשרת ללקוחותיה לנווה את כל הלידים בצורה אפקטיבית יותר, ניהול כל התקשרות מול לקוחות הקצה שלהם, ניהול כספים והעברת תשלוםם, חתימת חוזים, תזמון פגישות, ניהול משימות, אוטומציה וכו'.

הפלטפורמה עוזרת יומיום לעשרות אלפי אנשים בארץ"ב להתנהל בצורה אפקטיבית ומקצועית יותר, כך שהם סוגרים יותר עסקאות בפחות זמן ומאמץ. את הזמן הפנוי שלהם הם יכולים השקיע בהגדלת העסק, מציאת עוד לקוחות ובמשפחה שלהם.

בהאניבוק הלקוח הוא המרכז ואיתו גם הbranding שלו. חשוב לנו לוודא שאנחנו מאפשרים לו להראות כיצד טוב שהוא יכול בקשרו של הלקוחות שלו. עם טכנולוגיות מתקדמות ודגש על עיצוב, אנחנו מאפשרים לו בקלות לבנות חוזים, הצעות מחיר ואי מיילים שנראים טוב ומאפשרים תקשורת מהירה ויעילה עם הלקוחות שלו.

אנחנו מתמודדים עם אתגרים טכנולוגיים, עיצובים ופיננסיים. כאשר בכלאתגר אנחנו שמים את הלקוח במרכז על מנת להגיע להחלטה נכונה ומהירה. בואו תצטרפו לחברת מצלחה שרצה לשנות את הדרך בה עצמאיים עושים עסקים. חברות שמאפשרות ללקוחותיה להתפרק מהחלום שלהם.

וכבר הזכרנו שהופענו בראשית מקומות העבודה האטרקטיביים ביותר לשנת 2018 ו 2019? גם בישראל וגם בסן פרנסיסקו (אם תהיתם).

אז למה לעבוד בהאניבוק? כי התרבות העבודה פה מדהימה. כי كيف הגיעו כל בוקר לעבוד עם אנשים מוכשרים כלכך. כי כל יום שומעים מאות פידבקים מדהימים מליקחות שינוים להם את החיים. כי האתגר הטכנולוגי דוחף אותנו כל יום לבנות דברים חדשים ולשפר את מה שכבר בנוינו.

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

על ריאקט

צד הלוקו הוא הכנוי לקבצים ששרת האינטרנט שלוח אל המשתמש והם-הם בעצם אתר האינטרנט. בדרך כלל מדובר בקובץ אחד או יותר של HTML ו-CSS, בקובצי תמונות ובקובצי ג'אווהסקריפט. הדףן יודיע לקרוא את כל הקבצים האלו ולבנות מהם תמונה של אתר, ה-HTML קובע את מבנה האתר, ה-CSS והתמונות קובעים את העיצוב שלו וקובצי הג'אווהסקריפט את התנהוגותו. בתחלת ימי הרשת כך נראה אתר אינטרנט; בתחילת הג'אווהסקריפט שימושה לאנימציות או לאינדיקציות שונות בדף ובהמשך לתקדים מתוכנים יותר כמו שליחת בקשות באמצעות AJAX. ובכל זאת, באתר האינטרנט הראשוניים, כל אינטרנטאקטיב ניוטו כלהי – לחיצה על כפתור בתפריט או שיגור טופס – שירה בקשה לשרת וגרמה לטעינה מחודשת שלו בידי הדףן ולקבלת סט חדש של קובצי CSS, HTML וג'אווהסקריפט.

במהלך הזמן החלו להפתח ספריות ג'אווהסקריפט, כמו jQuery. ספריית jQuery הייתה ספרית עזר שסייעת למפתחני ג'אווהסקריפט ליצור אפקטים ואנימציות בклות הרבה יותר. קוד הג'אווהסקריפט הצד הלוקו הפך להיות שימושי יותר ויותר. בשנת 2008 יצא לשוק ספריה שנקרה Backbone.js. זו הייתה ספריה ששינתה彻底 את הדרך שבה אנו משתמשים לצד הלוקו: במקומ קוד שמאגדיר רק התנהוגות – אפליקציה שלמה שיושבת לצד הלוקו ומתחנגת כמו אפליקציית צד שרת, כולל אפשרויות ניוט בעמודים, הباتת מידע מ-API של שירותים ועוד שימושים רבים. היו שימוש בספריות גדולות כלו, או יותר נכון פרימורוקים המאגדירים את התנהוגות לצד הלוקו, היו רבים – מילוט פיתוח ועד חווית שימוש יוצאת דופן עבור הלוקו. הפרימורוקים האלה אפשרו ליצור SPA – Single Page Application. כמובן, כשלישים באתר ובמעבר בין דפים אין טעינה מחדש מהשרת אלא מעבר חלק בין דף לדף באמצעות רכיב מבוסס ג'אווהסקריפט שנקרא רואטר (ועלוי נלמד בהמשך הספר). Backbone.js הייתה הראשונה, אבל מהר מאוד הגיעו ספריות נוספות כמו Ember.js ו Ember.js כמודול אנגלול. ריאקט הגיעו אחרי אנגלול וגרסתה הראשונה יצאה ביוני 2013. מאז היא תפסה תאוצה משמעותית והפכה לאחת מהספריות הגדולות והנפוצות בעולם.

ריאקט (React) היא ספרייה מבוססת ג'אווהסקריפט המיועדת לצד הלוקו. היא מאפשרת לנו ליצור אתרים שלמים ומערכות שלמות בклות הרבה ובדרכן מודרנית ופושטה. באמצעות ריאקט אנו יכולים ליצור דפי אינטרנט או אפליקציות שרצות על טלפונים ניידים ואפילו על מחשבים בклות הרבה.

במקור, המפתחת של ריאקט היא חברת פיסבוק, שעדי היום היא התומכת הראשית שלה והפתחים שלה מוביילים את פיתוח הליבה של ריאקט ומתוים את הדרך. ריאקט מפותחת בראשון קוד פתוח מלא (החל מגרסת 16) וקוד המקור שלה נמצא בGITהאב. אפשר להשתמש בה לכל שימוש בצוורה חופשית. אחד היתרונות הגדולים בראקט הוא עשר הקומפוננטות שימושות בה, מה שאומר שאפשר ליצור בклות אפליקציות מורכבות באמצעות הקומפוננטות שפותחים אחרים פיתחו – דבר המאפשר לכל צוות פיתוח שבוחר בראקט גמישות ועובדת מהירה מאוד. גם סביבת הפיתוח של ריאקט וכי הבדיקות שלה, החינויים לפיתוח בקנה מידה גדול, הם מצוינים ועמידים מאוד. יש להCommerce גם חסרונות – החיסרון העיקרי הוא שריאקט לא קובעת עבור המפתח את הרכיבים שאיתם 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 המתכנתים שצרכים לפתח אתר כלשהו בצד הלוקוט בוחרים בריאקט.

הkonosfzia של ריאקט ודרכו העבודה בה לא שונות מהותית מספריות אחרות כמו אנגלר או Vue. אך אם איןכם מכירים אף פריימוורק או ספרייה של ג'אווהסקריפט, ריאקט היא מקום מצוי להתחילה בו את המסע לעולם המופלא של פיתוח צד לקוח. זאת אף שריאקט שונה בכמה דרכי מהותיות, שאוינו נלמד בהמשך, מספריות אחרות כמו אנגלר.

דרך הלימוד

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

יכול להיות שלמרות ההסבירם ולמרות הדוגמאות לא תבינו נושא מסוים או שלא תבינו אותו עד הסוף, עמוק. זה קורה לטובים ולمبرיקים ביותר. הפתרון? חיפוש בגוגל – במיוחד באנגלית. כיוון שריאקט היא כל כך פופולרית, יש סיכוי סביר כי יותר שמשהו כבר נתקל בעניה הזה וכותב עליה משהו. אטרים כמו StackOverflow והפורומים השונים מכילים שפע של מידע ותשובות לשאלות שונות. נוסף על כן, בפייסבוק יש לא מעט קבוצות מקצועיות בעברית שישמשו לשיער לכם – בפרק הסיכום של הספר יש כמה קישורים רלוונטיים.

על המונחים בעברית

אני כותב בעברית על טכנולוגיה ועל תכונות שכבר יותר מעשר, והדילמה באילו מונחים בעברית להשתמש מלווה אותי 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 לידיון העבודה ותגידו: "במפגש המתכנתים האחרון שמעתי על דרך חדשה לביצוע הידור שבודק הוצאות במנשך מבוסס הבדיקות", סביר להניח שלא תקבלו את העבודה. אבל אם תגידו, "בmitap האخرון שמעתי על דרך חדשה לביצוע קמפול שבודק אינדנטציה ב-API מבוסס פרומיסים" – יבינו על מה אתם מדברים. זו הסיבה שלא תמצאו בספר מילים כמו "הידור", "מחלקה" או "מרשתת", אלא "קמפול", "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 להכניסו אותו לחיפושים שלכם בגוגל.

חשוב לציין שאיני בז כל לאקדמיה ללשון וshall ממה מונחים שלא אכן נכנסו לשפה המדוברת במרכז הטכנולוגיה השונים (למשל: קובץ או מסד נתונים), אבל בכל מקום שהייתה ביד' הבחירה בין להיות מובן לבין לעמוד בכללי הלשון, העדפת להיות מובן.

סביבת העבודה הבסיסית

זה הפרק החשוב ביותר, כיוון שאי-אפשר ללמידה קוד בלי לכלוך את הידיים בקוד משלכם. קריטי לקרוא את הפרק הזה וליצור סביבת עבודה בסיסית על המחשב שלכם. אם אתם נתקלים בקושי כאן – אל תוותו ואל תתייחסו. התקנת סביבת העבודה היא החלק הקשה ביותר בלימוד טכנולוגיה חדשה. נסו שוב ושוב – בצעו ריסטרט למחשב, נסו מחשב אחר, שדרגו את מערכת הפעלה, נסו מדפסן אחר שאינו הדפסן המקורי שלכם – כל טכניתה וטכניתה. פשוט אי-אפשר לדלג על השלב זהה.

ריאקט מורכבת מכמה חלקים –/column בג'אוהסקרייפט. החלק הראשון הוא הספרייה עצמה: ריאקט. מדובר בקובץ ג'אוהסקרייפט מרכזי המכיל את כל הפונקציונליות של הספרייה. הקובץ מכיל מודולים של ג'אוהסקרייפט וגם משתנים גלובליים ואחרים – ובלעדיו אי-אפשר להשתמש בריאקט. כשאנו בונים סביבת עבודה בסיסית, אנו נזקקים לריאקט. הגיוני, לא? החלק השני הוא `dom-react`. גם הוא חלק מהספרייה וגם הוא כתוב בג'אוהסקרייפט. הוא מכיל את הפונקציות הקשורות בין ריאקט-LDOM. ראשית התיבות של DOM הם Document Object Model ואני ארחיב לגביו בהמשך. כרגע ציריך פשטוט לזכור שמדובר בעוד חלק של הפרויימורק שהוא צורך איתנו כשהאנחנו מפתחים עבור אתר אינטרנט.

החלק השלישי הוא `babel`. מה זה? מדובר בספריית ג'אוהסקרייפט שימושית ופופולרית מאוד. יש לה כמה תפקידים חשובים. במקור babel סיעה למתכנתים שרצוי לתאים את קוד הג'אוהסקרייפט שלהם לדפינים שונים שלא תמכדו בפתרונות החדשניים של השפה, למשל דפינים שלא ידעו מה זה או `let` או `const`. הספרייה הזאת לוקחת את כל הקוד המודרני של ג'אוהסקרייפט והעבירה אותו תהליך, שבמסגרתו הוא הפרק לקוד שתואם גם דפינים אחרים. למשל, היא מירה את `let` ל-`var`, כך שדף ישן יוכל לעבוד אליו. התהליך הזה נקרא "טרנספירציה" – זו המילה המדעית המחשב שמשמעה ללקחת קוד כתוב בשפה מסוימת ותרגמו אותו לקוד כתוב בשפה אחרת. במקרה הזה ללקחת קוד כתוב בג'אוהסקרייפט מודרנית ולהעביר אותו לקוד כתוב בג'אוהסקרייפט מגרסאות קודומות.

במקרה שלנו, ל-`babel` יש תפקיד משמעותי בהמרת ה-`XJS` שלנו, שהוא הסינטקס שבו אנו כותבים בריект קומפוננטות לקוד HTML שהדף יודע לעבוד איתו. על `XJS` למד בהרחבה בהמשך. אלו החלקים הבסיסיים שחייבים להיות בסביבת העבודה שלנו. בנוסף על הקוד שלנו, אנו צריכים ליצור דף HTML ריק לחלוטין שקורא באמצעות `src` לשלוות הקבצים האלה וגם לקובץ שבו אנו כותבים את הקוד שלנו.

אנו יכולים להוריד את הקבצים האלה מהאתר של ריאקט או להשתמש בהם 'שירות מ-CDN'. ראשית 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 ספריות גדולות ומובן שריект, הפרויימורק הפופולרי בעולם, נמצאת ביניהם. ה-`CDN` שהדוקומנטציה של ריאקט משתמש בו הוא `pkg` וaned נשתמש בו. pkg מזינה את הקבצים בקישורים הבאים:

קובץ הפרויימורק של ריאקט:

<https://cdnjs.cloudflare.com/ajax/libs/react/17.0.0/umd/react.development.js>

שיםו לב: הספר מלמד על גרסה 17 של ריאקט ולכן קיימם המספר הזה ב-URL.

:react-dom

<https://cdnjs.cloudflare.com/ajax/libs/react-dom/17.0.0/umd/react-dom.development.js>

:babel-h-

<https://unpkg.com/babel-standalone@6.26.0/babel.min.js>

כך קובץ HTML שלנו נראה:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>React development environment</title>
<script crossorigin
src="https://unpkg.com/react@17/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js"></script>
<script crossorigin src="https://unpkg.com/babel-
standalone@6.26.0/babel.min.js"></script>
</head>
<body>
<div id="content"></div>
</body>
<script type="text/babel">
console.log('Here will be React code');
</script>
</html>
```

בתחתיית הקובץ נמצא הקוד שלנו. אלו מסומנים ל-babel שהוא קוד שצריך לעבור טרנספילציה באמצעות הצמדה של:

`type="text/babel"`

לtagית `script`. בין התגיות הפתוחות של `script` לתגית הסוגרת שלו, אנו יכולים לכתוב את הקוד הבא:

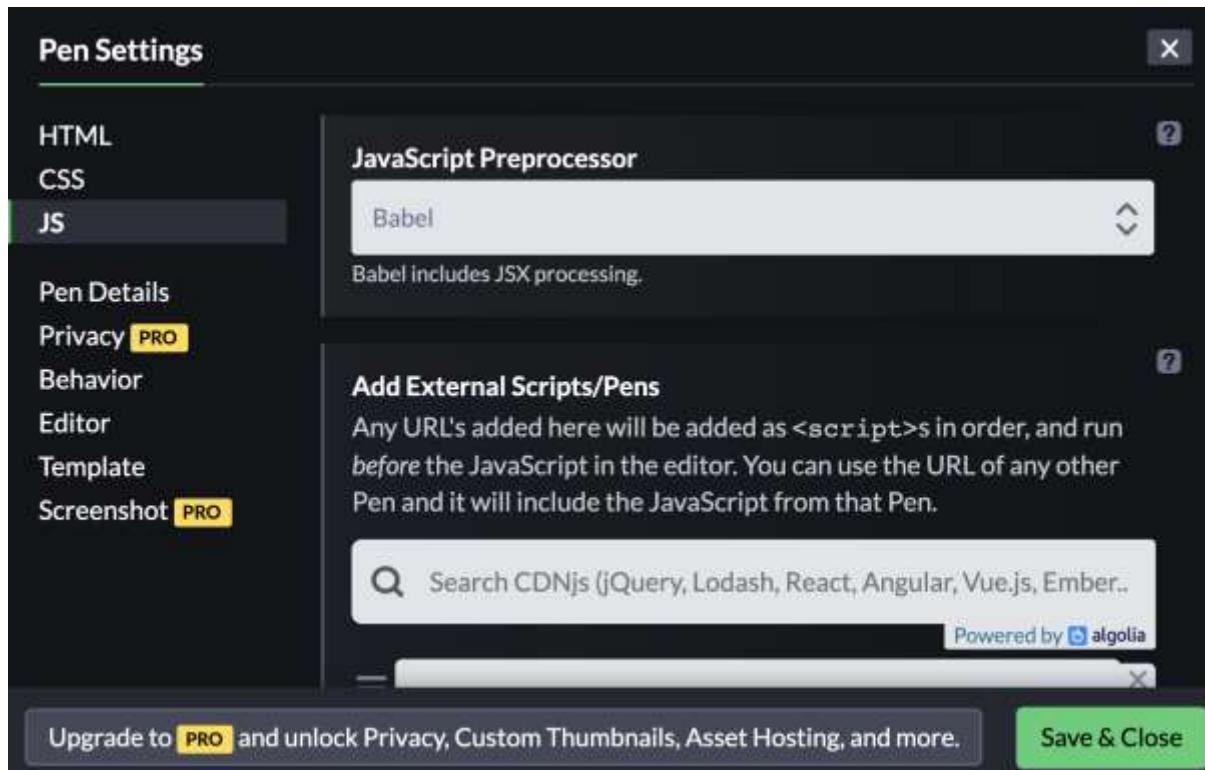
`console.log('Here will be React code');`

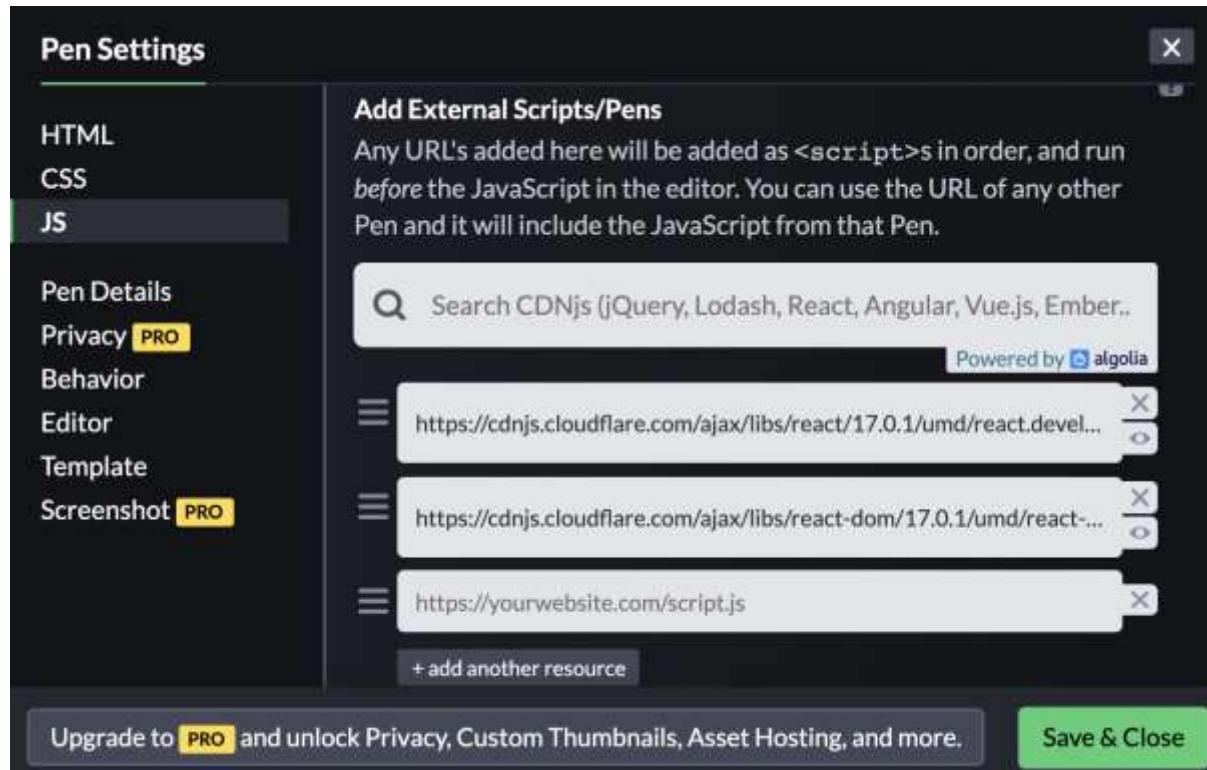
באמצעות עורך הקוד החביב عليכם (כמו Visual Studio Code החינמי), צרו את קובץ HTML זהה ושמרו אותו במחשב המקומי שלכם. פתחו את קובץ HTML בדף באמצעות כניסה לתיקייה שבה הקובץ נשמר ולהיכזה על "פתח עם דף"¹ ידאו שהוא עולה וشبוקונסולה של כל המפתחים מופיע `Here will be React code`. אם כן – זהו, אנחנו מוכנים לעבודה ראשונית.

סביבת עבודה בסיסית ב-codepen

אפשרות נוספת היא לבנות סביבת עבודה מרוחקת, שאם בה אפשר לכתוב ג'אווהסקריפט ולצפות בתוצאות באמצעות דפדן בלבד. סביבת העבודה זו זמינה בחינם בכמה וכמה כלים ואתרים, אבל האתר הכי פופולרי ומוצלח הוא codepen.io. מדובר באתר שמאפשר לכולם (אפילו בלי רישום, אף על פי שמומלץ להירשם כי זה מאפשר שמירה של הקוד שלכם) לכתוב קוד פשוט בג'אווהסקריפט, HTML ו-CSS. העבודה באתר פשוטה: נכנסים אל <https://codepen.io/pen/>, מתחילהם לכתוב קוד וראים את התוצאות.

כדי לעבוד עם codepen וריאקט חיבים להכניס את שלושת קובצי הג'אווהסקריפט שהזכרתי קודם: קובץ הפרויימורק של ריאקט, קובץ ה-react-dom וקובץ ה-babel. איך עושים את זה? בהגדרת כל "פרויקט" (שנקרא חק באוטו אטר) הקפידו להכניס את babel כ-preprocessor וקישור ידני אל קובצי הג'אווהסקריפט מה-CDN. זה נראה כך:





עשיתי את זה עבורכם בפרויקט משלי 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

חשוב לציין שגם סביבת פיתוח בסיסית מאוד של ריאקט ושהיא כמובן מוכוונת פיתוח בלבד ומיעדרת ללימוד, אבל זה אמרו להספיק לכטיבת אפליקציית הריאקט שלנו והקומפוננטה הראשונה.

פרק 1

אפולו הצעית וריאקט



אפליקציית ריאקט

כל הקוד של ריאקט אמרור להיות בתוך אפליקציית ריאקט. זה סוג של מתחם שבו הקוד מבוסס הריאקט שלנו עובד. בעצם, מדובר באלמנט אב של DOM שמתוחתיו ריאקט שליטה, יוצרת DOM משלה, שנקרא Virtual DOM, וחייה. יכולות להיות כמה אפליקציות ריאקט בדף HTML אחד. מבן שכל אחת מהן תהיה מתחת אלמנט DOM אחר. אפליקציית הריאקט היא בעצם אלמנט DOM שאנו מכירים עליינו כשלנו, ובו אנו יכולים ליצור את המרכיבים של האתר שלנו – שהם קומפוננטות הריאקט. כדי ליצור אפליקציה ריאקט מתחת אלמנט מסוים אנו חייבים פשוט... לבחור אותו. אנו נגידיר div, שהוא אלמנט HTML פשוט ביותר עם id פשוט. להזכירם – id הוא סלקטור ייחודי המגדיר אלמנטים ב-HTML שאנו בוחרים לחת להם זהות ספציפית. בקיצור ה-HTML שלנו כבר יש הגדרה של div כזה:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@17.0.1/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@17.0.1/umd/react-dom.development.js"></script>
  <script crossorigin src="https://unpkg.com/babel-standalone@6.26.0/babel.min.js"></script>
  <script src="local-react-code.js"></script>
</head>

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

</html>
```

הינה ההגדרה – div שה-id שלו הוא :content

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

זה המקום שבו תציב אפליקציית הריאקט שלו. אנו צריכים רק לזכור שה-`p` הוא `.content`. את אפליקציית הריאקט אנו יוצרים באמצעות:

ReactDOM.render

המוגדר הזה בא עם האובייקט הגלובלי ReactDOM. האובייקט הגלובלי זהה בא בזכות קובץ ה-`react-dom.js`, שדגנו שהוא בסביבת הפיתוח שלנו. המוגדר מקבלת שני ארגומנטים: אחד הוא JSX (שעוד מעט נלמד בדיק מה הוא), והשני הוא האלמנט שאנו בוחרים כדי להריץ בו את ה-JSX זהה.

הציבו את הקוד הזה בין תגיות הסקריפט שלכם:

```
<script type="text/babel">
const target = document.getElementById('content');
ReactDOM.render(
<h1>Hello World!</h1>,
target
);
</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 שהעתיקתם את הקוד כשרה והמשיכו רק כאשר אתם יודעים שסביבת הקוד שלכם עובדת.

הבה נعبر על הארגומנטים השונים של ReactDOM. נתחילה מהשני, ה-target. אני יוצר רפנסו לאלמנט שבו אני רוצה להציב את התוכן שלי באמצעות:

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

אני מעביר אותו לקבוע target כארגומנט השני. אתם אמרוים להכיר את זה אם יש לכם ידע בג'אוויסקሪיפט. אם לא, אני חזרו על החומר של ג'אוויסקሪיפט ו-HTML (שנמצא גם בספר שלי "לימוד ג'אוויסקሪיפט בעברית").

עכשו נדבר על הארגומנט הראשון:

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

הוא לא מורכב מדי', בסך הכל תגיית HTML. אבל שימו לב למשהו מעניין – ה-HTML הזה נמצא בתוך קוד ג'אוויסקሪיפט! איך זה יכול להיות? הרי אם תשתמשו בקוד HTML ללא מירכאות בקוד ג'אוויסקሪיפט, הקוד ידפיס שגיאה ויפסיק לפעול. ג'אוויסקሪיפט לא מכירה HTML ולא יודעת לעבד אליו. איך יכול להיות שאני משתמש ב-`h1` וב-HTML בג'אוויסקሪיפט ללא מירכאות והקוד לא נופל?

הסבירה היא JSX. זוכרים את המרכיב החלישי בסביבת הפיתוח שלנו, ה-`babel`? הוא אחראי למצוא את כל התגיית של HTML שיש בקוד הג'אוויסקሪיפט ולהמיר אותה למשהו שג'אוויסקሪיפט יודעת להתמודד איתה. הקוד הזה, שכרגע יש בו HTML פשוט בלבד, הוא JSX – ראיי תיבות של XML JavaScript – והוא אחד מהפתרונות 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'אוויסקሪיפט לכל דבר ועניין. אני יכול לשים אותו במשתנה לצורך העניין:

```
const target = document.getElementById('content');

const value = <h1>Hello World!</h1>;
ReactDOM.render(
  value,
  target
);
```

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

תרגיל

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

פתרון

```
<!DOCTYPE html>
<html>
```

```

<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@17.0.1/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-
dom@17.0.1/umd/react-dom.development.js"></script>
  <script crossorigin src="https://unpkg.com/babel-
standalone@6.26.0/babel.min.js"></script>
</head>

<body>
  <div id="my-react-app"></div>
</body>
<script type="text/babel">
const target = document.getElementById('my-react-app');
const value = <h2>I am learning React!</h2>;
ReactDOM.render(
  value,
  target
);
</script>

</html>

```

ראשית ניצור קובץ HTML פשוט שבו יש קריאה לשלוות הקבצים שאנו צריכים לאפליקציית ריאקט וועליהם הסבירנו בתחילת הפרק.

הצעד הנוסף הוא ליצור div שיש לו id מסוים שאנו רוצים להכניס אליו את אפליקציית הריאקט. במקרה זה:

```

<div id="my-react-app"></div>
השלב הבא הוא ליצור את אפליקציית הריאקט באמצעות:
ReactDOM.render

```

המתודה זו מקבלת שני ארגומנטים: האחד הוא ה-XJS והשני הוא האלמנט שהאפליקציה תהיה בו. הארגומנט הראשון הוא XJS פשוט שאינו מ-HTML, מלבד העובדה שהוא פשוט בטור קובץ ג'אויסקcript:

```
const value = <h2>I am learning React!</h2>;
```

הארוגמנט השני הוא האלמנט שהאפליקציה תהיה בו. אני מקבל אותו באמצעות `getElementById`, שהוא מתודת ג'אויסקcript שנמצאת בדף באופן טבעי (בלי קשר לריאקט), ועביר אותו באמצעות משתנה:

```
const target = document.getElementById('my-react-app');
```

אני מכניס את שני המשתנים ל-`render` לפי הסדר, ראשית ה-XJS ו שנית המטרה שלי:
זה הכל:

```
ReactDOM.render(  
  value,  
  target  
)
```

שמירת ה-HTML ופтиיה שלו באמצעות הדף תציג לי!

פרק 2

בנייה קומפוננטת פונhaziיה ובסביבת העבודה הבסיסית



בנייה קומפוננטת פונקציית בסביבת העבודה הבסיסית

אחרי שלמדנו לבנות את האפליקציה ואיפלו למדנו על JSX בסיסי, הגיע הזמן ללמידה על המבנה הבסיסי של ריאקט – קומפוננטות. אחת המהפקות הגדולות בפיתוח צד לקוח שריект הביאה היא הקומפוננטות. מדובר ביחידת תוכנה קטנה שאפשר להשתמש בהן שוב ושוב במקומות שונים באתר שלנו. המתכניםים בריאקט יוצרים למשל קומפוננטה אחת של טבלה הנינתנת למילון ויכולים להשתמש בה בכל מקום באפליקציה או באתר שלהם. אם הם צריכים לשנות את הטבלה, הם משנים קומפוננטה אחת בלבד. בכל אפליקציית ריאקט יכולות להיות אינספור קומפוננטות. למתכנים חדשים קל לחשב על קומפוננטות בתור פונקציות.

בתרגיל בפרק הקודם יצרנו אפליקציית ריאקט שבה כתוב "I am learning React!". אם אני רוצה לכתוב כמה פעמים "I am learning React!" או, אני יכול לשכפל את ה-h2 ב-`h2`-JSX באופן הבא:

```
const target = document.getElementById('my-react-app');

ReactDOM.render(
  <div>
    <h2>I am learning React!</h2>
    <h2>I am learning React!</h2>
    <h2>I am learning React!</h2>
  </div>,
  target
);
```

שימוש שמי לב שאני צריך לעטוף את השכפולים שלי ב-`h2` אחד מكيف, כיוון שפונקציית `render` דורשת ממני 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 שתתדייס עבורנו את:

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

יש שני סוגי קומפוננטות – סוג אחד הוא קומפוננטות מבוססות **קלאסו**, שעליהן נלמד בהמשך הספר. הסוג השני והנפוץ יותר הוא קומפוננטה של פונקציה, והיא פשוטה למדוי. מגדירים אותה באמצעות פונקציה (זו לא הפעעה גדולה, נכון?). כרגע זה די פשוט. הקומפוננטה נראהthus כר:

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

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

```
<Greeting />
```

אלמנט שסגור את עצמו. או כר:

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

אף על פי שהkonvencija החד-משמעות היא לכתוב אלמנט שסגור את עצמו.

ואיך הקוד המלא שלנו יראה? כך:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@17.0.1/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-
dom@17.0.1/umd/react-dom.development.js"></script>
  <script crossorigin src="https://unpkg.com/babel-
standalone@6.26.0/babel.min.js"></script>
</head>

<body>
  <div id="my-react-app"></div>
</body>
<script type="text/babel">

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

const target = document.getElementById('my-react-app');
ReactDOM.render(
  <div>
    <Greeting />
  </div>,
  target
);
</script>
</html>
```

אני יכול לשכפל את הקומפוננטה כרצוני, כמוובן:

```
function Greeting() {  
  return <h2>I am learning React!</h2>;  
}  
  
const target = document.getElementById('my-react-app');  
ReactDOM.render(  
  <div>  
    <Greeting />  
    <Greeting />  
    <Greeting />  
    <Greeting />  
    <Greeting />  
  </div>,  
  target  
);
```

היתרון הגדול? אם אני רוצה לשנות את הכתיבה בקומפוננטה או את האלמנט או להוסיף לקומפוננטה, אני יכול לעשות את זה במקום אחד בלבד, ובבהת אחת השינוי הזה ישפיע על כל שימוש ושימוש בקומפוננטה בתחום האפליקציה שלי.

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

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

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

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

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 – הישר באפליקציה או בתוך כל קומפוננטה אחרת. במקרה זהה מי משתמש בה הוא קומפוננטת Greeting. איך היא משתמשת בה? בדיק כmo ב-HTML. שם הקומפוננטה כתגית HTML:

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

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

אני יכול להשתמש, כמובן, בקומפוננטת Greeting כמה פעמים שאני רוצה. בכל פעם שאני משתמש בה, אני אראה על המסך Hello, I am learning React!. אם-arצה לשנות את הברכה, יוכל לעשות זאת בקלות דרך שינוי מקום אחד. זה מה שיפה בקומפוננטות ריאקטיביות – אפשר לבצע בהן שימוש חוזר (המונח המקובל הוא reuse) בכל מקום. קומפוננטות שמכילות עוד קומפוננטות ובתוכן יש עוד קומפוננטות וכך הלאה; הכל מתנתק בסופו של דבר לאפליקציית ריאקט אחת שמכילה בתוכה אינספור קומפוננטות.

תרגיל:

צרו אפליקציה ריאקט שבתוכה יש קומפוננטה אחת שנקראת Root. בקומפוננטת Root יש שתי קומפוננטות נוספות, אחת שנקראת Soigo ומחזירה JSX JS שהוא .Prepare to die!, והשנייה שנקראת Greeting ומחזירה JSX JS שהוא!Hello, 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>
}

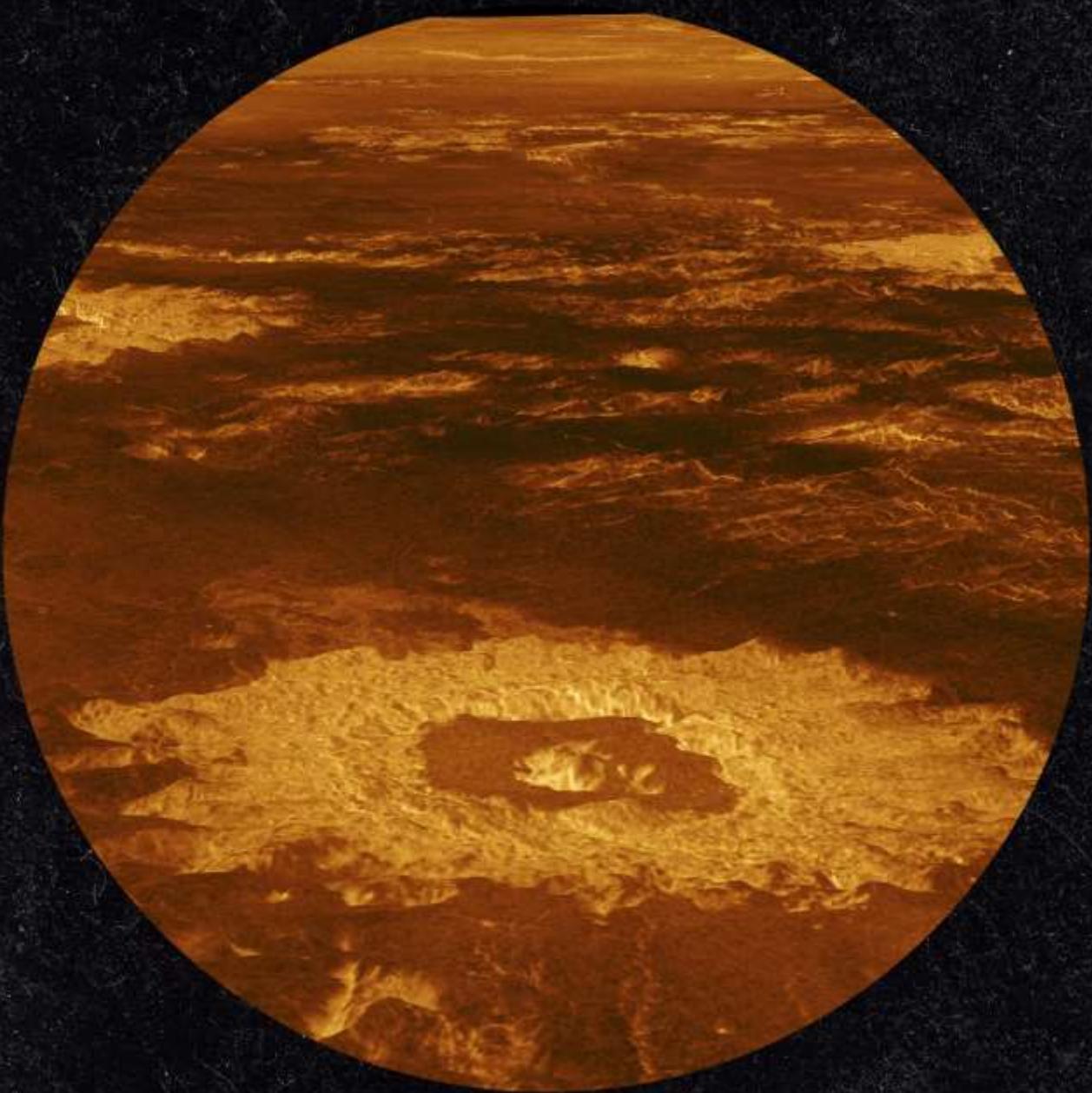
const target = document.getElementById('my-react-app');
ReactDOM.render(
  <div>
    <Root />
  </div>,
  target
);
```

יצרת שולש קומפוננטות פשוטות. הראשונה והשנייה Soigo ו-Greeting מוחזירות JSX פשוט. כדי לשים לב שב-XJS אני חייב לשים אלמנט אב אחד, במקרה זהה בחרתי ב-span. הקומפוננטות הללו הן קומפוננטות מסווג פונקציה. כדי לשים לב שמן חייב להתחיל באות גדולה. מהרגע שהגדרתי אותן, אני יכול להשתמש בהן בכל קומפוננטה אחרת בדיק כmo 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

בניה של סביבת REACT אמיתית ומודרנבת יותר CREATE REACT APP באנצטול



בניה של סביבת ריאקט אמיתית ומורכבת יוטר באמצעות Create React App

בפרקים הקודמים למדנו להקים סביבת עבודה פשוטה וראינו איך יוצרים אתרים או אפליקציות ווב באמצעות אפליקציית ריאקט וקומפוננטות פשוטות. סביבת העבודה הזה היא פשוטה מאוד ומלמדת אותנו שריאקט זה לא קסם ולא וודו – אבל רובם המוחץ של המתכננים לא משתמשים בסביבת ריאקטazzo. היא פשוטה ופרימיטיבית מדי. כמה נוח היה אילו היו סביבה שלנו שרת מובנה, במקומם לטען את הקובץ דרך מערכת הקבצים המקומיית, או `hot reload`, שמר�新 אוטומטית את האפליקציה בסביבת הפיתוח בכל פעם שאנחנו עושים שינוי, או דיבאגר מובנה... כל אלו ועוד קיימים ממש מэнkapסיה בריект.

יחד עם ריאקט יש פרויקט שנקרא `Create React App`, המפותח ומתחזק על ידי הצוות של ריאקט. הפרויקט הזה הוא `Bootstrapper`. ככלומר הוא פרויקט של ריאקט עם המון תוספות שהוצאות של ריאקט חשב שהן טובות וחינויוות לפרוייקט חדש. זו בעצם סביבת פיתוח מלאה הכוללת שרת שאפשר להפעיל בקלות מכל מחשב. סביבת הפיתוח הזה מבוססת על `Node.js`, אך לא נדרש ידע ממשוני ב-`Node.js` על מנת לעבוד בה.

אנו לא יכולים להישאר בעולם ה-`Hello World`, ולימוד בניית סביבה אמיתית עם `Create React App` 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 ומישחו אחר כבר חוו כל תקלת שתחולו בדרך (אם תחולו). חיפוש מהיר של התקלה בגוגל יסייע לכם מאוד.

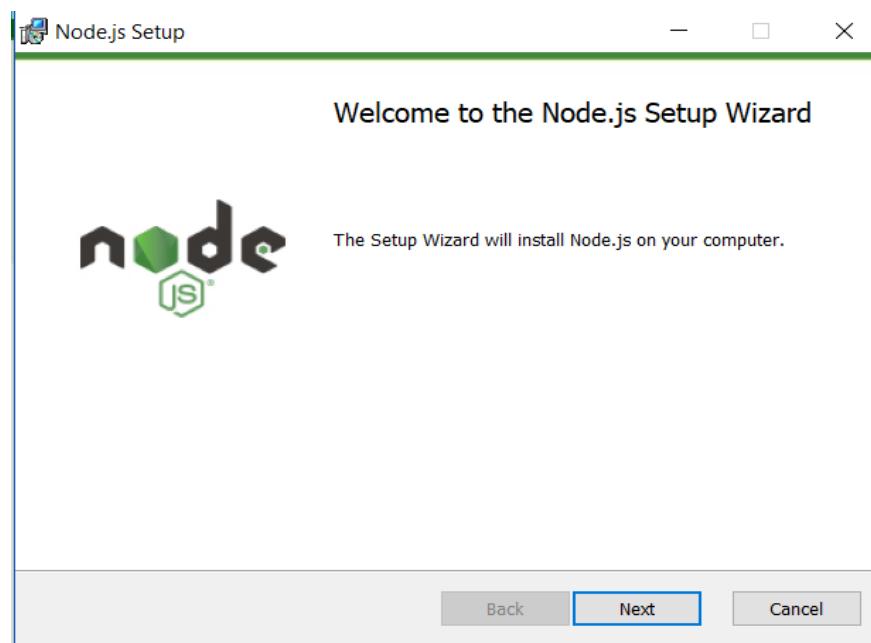
התקנת `Node.js` על המחשב שלכם

ראשית, אתם חייבים להתקין את `Node.js` על המחשב המקומי שלכם עובדים בו. `Node.js` הוא סביבה המאפשרת להריץ ג'אווהסקריפט על גבי מערכת הפעלה, במקרה הזה מערכת הפעלה 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/שרטים – ואת זה עושים באמצעות `Node.js`. הפלטפורמה הזה ניתנת להתקינה בכל מערכת הפעלה שהיא ובקלות. בחרו את מערכת הפעלה שלכם והתקינו את `Node.js` לפי ההוראות.

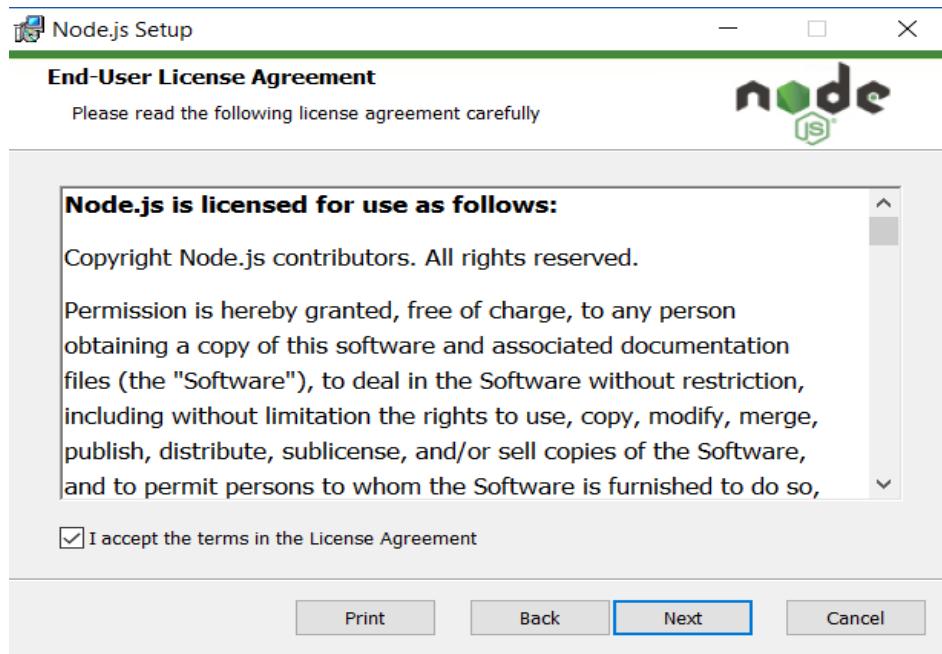
התקנה על חלונות

התקנה של `Node.js` על חלונות היא פשוטה. נclid בגוגל `Node.js Download` או נכנס אל: <https://nodejs.org/en/download/>

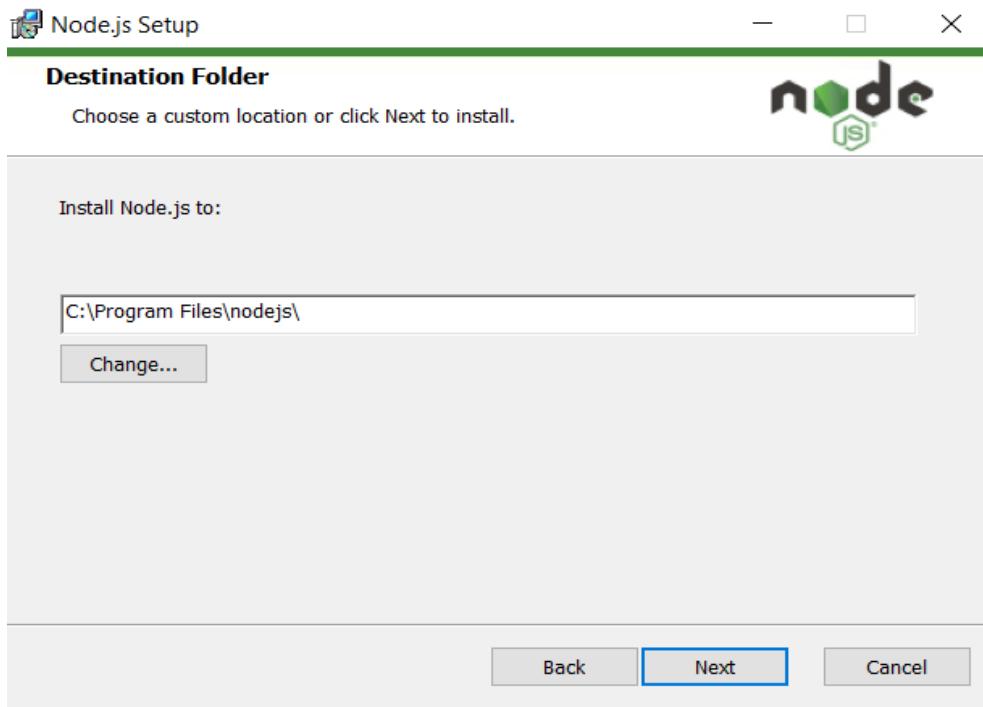
אנו נבחר בגרסת LTS – ראש תיבות של "గרסה לטווח ארוך", ונבחר במערכת הפעלה שלנו – אם מדובר בחלונות, יש לנו installer נוח. מורידים, לוחצים על התוכנה שיורדת:



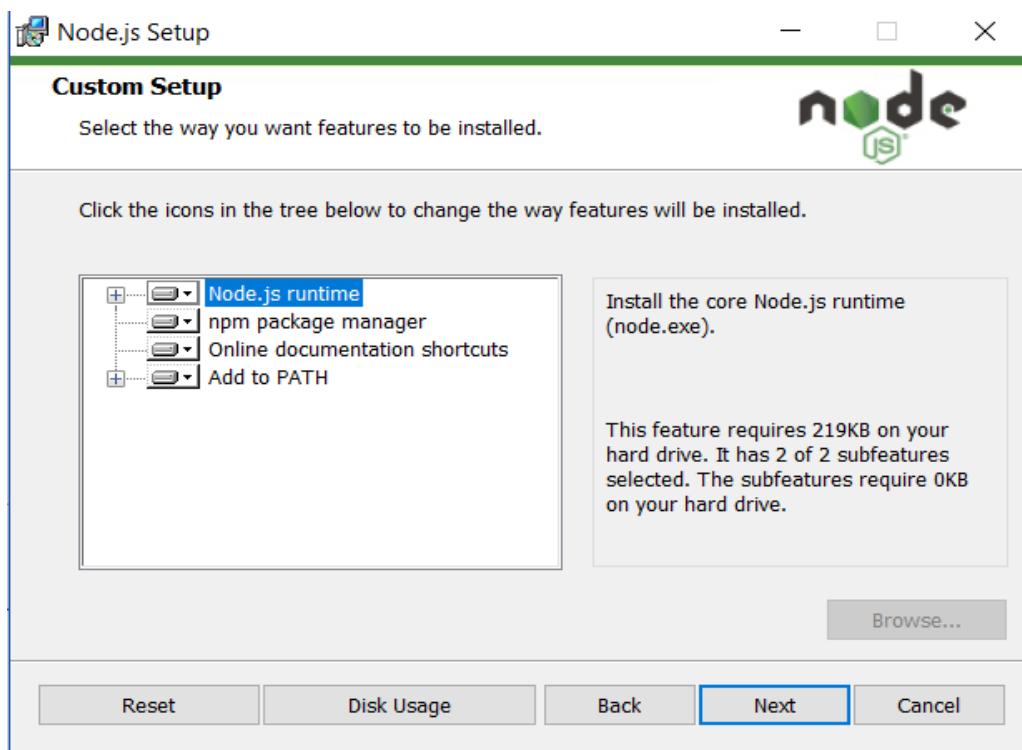
מקבלים את התנאים:



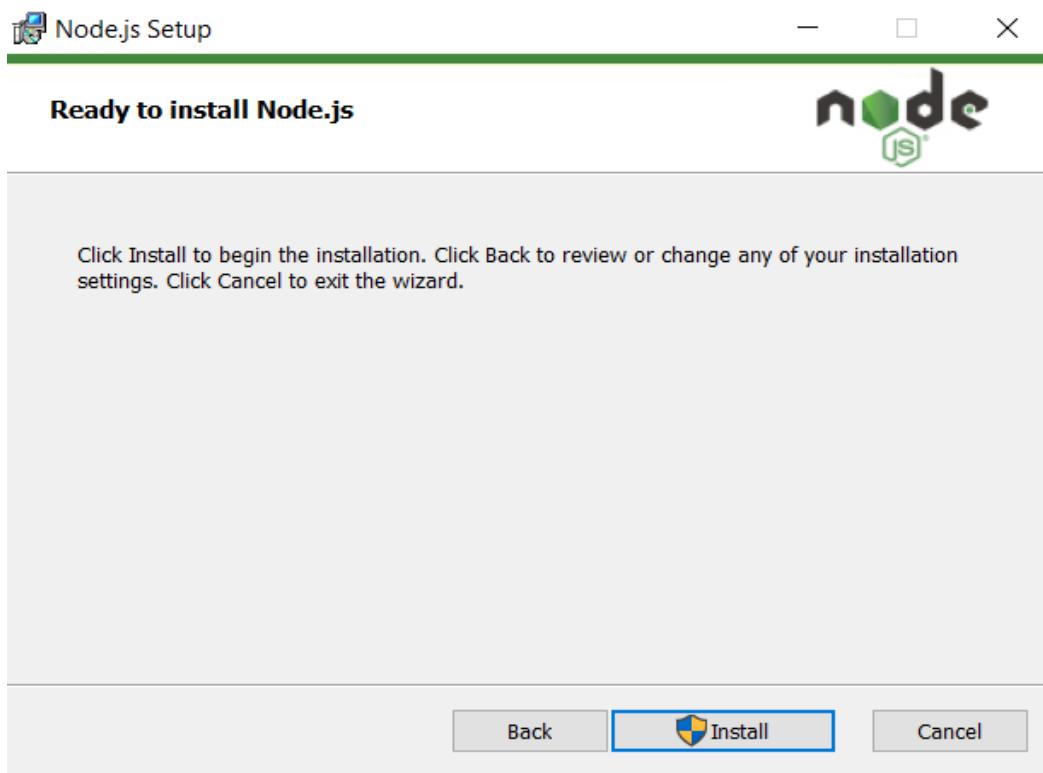
:Next לוחצים על



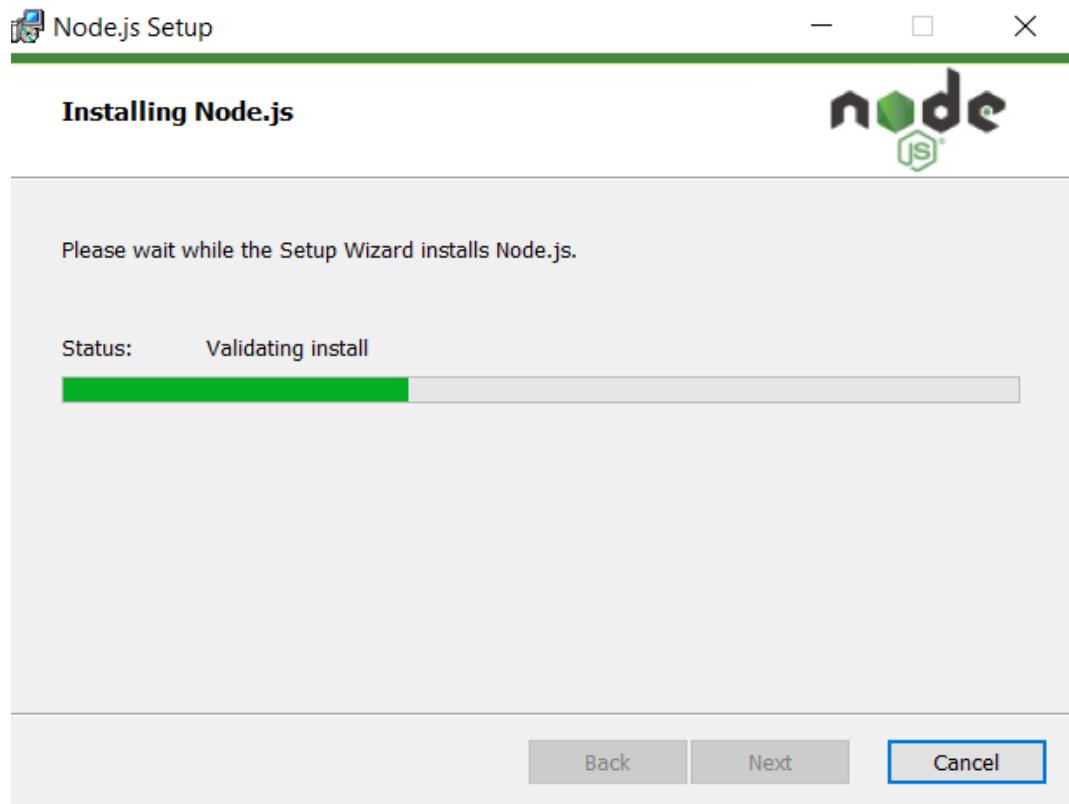
ושוב על :Next



לחיצה על Next לבסוף תתקין את התוכנה:



כל מה שנותר הוא ללחוץ לInstall בסוף ההתקנה:



התקנה על מק

ההתקנה של Node.js על מק פשוטה מאוד. נקליד בಗול Node.js Download או ניכנס אל: <https://nodejs.org/en/download/>

אנו נבחר בגרסת LTS – ראש תיבות של "גראסה לטווח ארוך", ונבחר במק – ירד קובץ dmg שהוא אפשר להתקין כמו כל תוכנה אחרת בהנחה שהמחשב שלכם הוא לא מחשב ארגוני שמנע התקנות מהאינטרנט. ההתקנה היא פשוטה ביותר.

אם אתם משתמשים ב-Zsh או ב-Oh My Zsh אני ממליץ להתקין את Node.js באמצעות homebrew באמצעות הפקודה `brew install node` (אם homebrew מותקנת אצלכם, כמובן שהוא יהיה מותקנת). קר או אחרת, לאחר ההתקנה, כניסה לטרמינל והקלדה של `-v` node יראו לכם את מספר הגרסה.

התקנה על לינוקס

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

אני יוצא מנקודת הנחה שמשתמשים בLINQPAD הם מיומנים בהרבה משתמשי חלונות וידיעם להתקין חבילת תוכנה ללא הסברים נוספים. כך או אחרת – לאחר ההתקנה, כניסה לטרמינל והקלדה של `- epoch` יראו לכם את מספר הגרסה בדיקן כמו במק.

עבודה עם טרמינל

כמו כל אפליקציית `Node.js`, גם `Create React App` מצריכה אותנו לעבוד עם טרמינל. טרמינל הוא המשך שבו אני מקlid פקודות למערכת הפעלה. כל מתכנת שעבד עם טרמינל כאשר הוא מתחבר מרוחק לשירותים שונים זהה ידע שימושי למדוי. אנו נלמד כיצד להתחבר לטרמינל ואיך לעבוד איתו בחלונות. אני לא מסביר על טרמינל במק או בLINQPAD כי אני יוצא מנקודת הנחה שמי שיש לו את מערכות הפעלה הללו יודע איך להשתמש באופן בסיסי בטרמינל.

הפעלת הטרמינל

בחלונות הגישה לטרמינל פשוטה למדוי. בחלונות 10 לוחצים על הזכוכית המגדלת, מקלידיים `cmd` ואז לוחצים על אונטרא.



מיד מקבלים מסך שחור. לווטיקים בינוינו הוא יזכיר את מסך ה-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 המשך הוא לא גרפי אלא טקסטואלי. יכול להיות שהזה ייראה לכם מישן ולא רלוונטי, בטח ובטע לאו ממשקים אחרים שאתם מכירים, אבל ככה רוב המתכנתים עובדים – מול הטרמינל של חלונות, של מק או של LINQPAD. חשוב מאוד להבין איך עובדים איתו.

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. אנטר בטרמינל הוא "שידור". אם תעשו את זה כמו שצרים, תראו משהו כזה:

```
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
10/21/2017  12:52 PM    <DIR>      .git
10/21/2017  12:52 PM    <DIR>      .gradle
10/21/2017  12:52 PM    <DIR>      .idea
10/21/2017  12:52 PM    <DIR>      .jre
10/21/2017  12:52 PM    <DIR>      .m2
10/21/2017  12:52 PM    <DIR>      .maven
10/21/2017  12:52 PM    <DIR>      .nodejs
10/21/2017  12:52 PM    <DIR>      .nuget
10/21/2017  12:52 PM    <DIR>      .oss
10/21/2017  12:52 PM    <DIR>      .sass-cache
10/21/2017  12:52 PM    <DIR>      .vs
10/21/2017  12:52 PM    <DIR>      .wsl
10/21/2017  12:52 PM    <DIR>      .yarncache
10/21/2017  12:52 PM    <DIR>      .zsh_history
```

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

	Name	Date modified	Type	Size
וות 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	
	.git	27/06/2017 12:52	File folder	

ניטוט בטרמינל

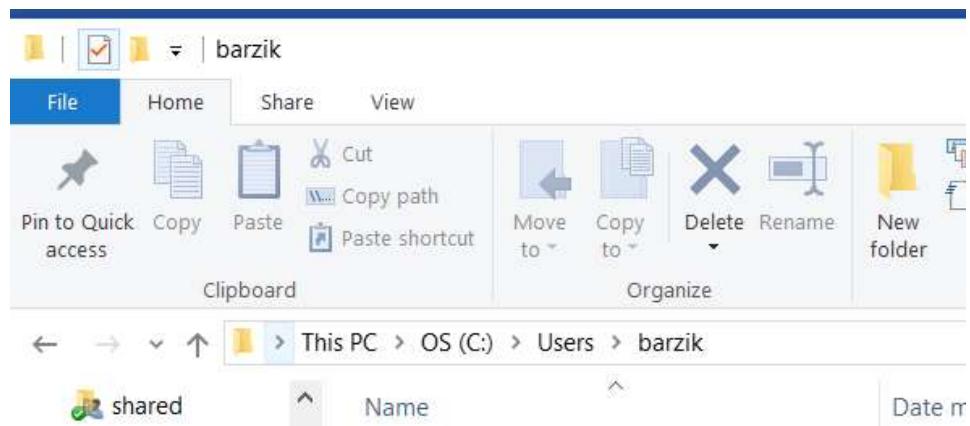
אתם יכולים לנוטו בטרמינל ולהגיע לתיקיות אחרות. למשל, אם יש לך תיקית Documents במיקום שלכם, אפשר להגיע אליה. נסו להקליד למשל: cd Documents. תגיעו לתיקיית Documents שיש תחת השם שלכם. אם תקלידו dir ותצפו בתוכן התיקייה, תראו שהיא זהה לתיקיית My Documents מסיר הקבצים. אפשר "לעלוות" לתיקייה אחת מעלה באמצעות .. cd. נסו לעלות שוב ושוב עד שתגיעו לתיקייה הראשית, הלווא היא: c. אנו יכולים לנוטו 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>
```

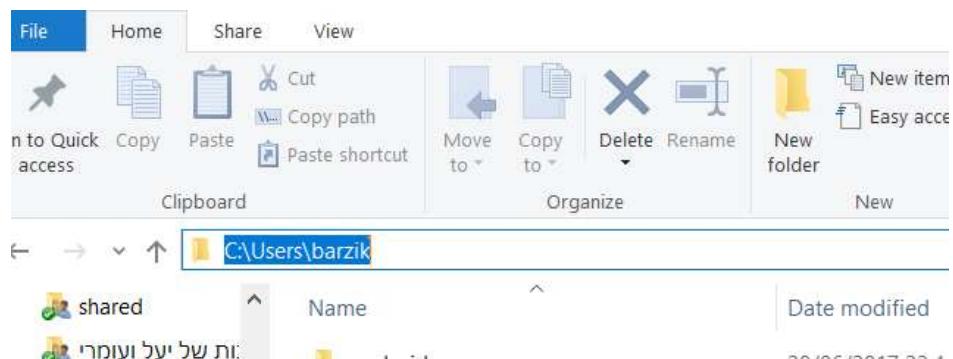
טיפ קטן – אפשר להקליד tab כדי לבצע השלמה.

מציאת מקומות בטרמינל דרך חלונות

אם אתם רוצים להגעת לתיקייה מסוימת ולא בטוחים מה מיקומה, מצאו אותה בסיר הקבצים הגרפי ולהוציא על הכותרת. למשל, התיקייה זו:



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.

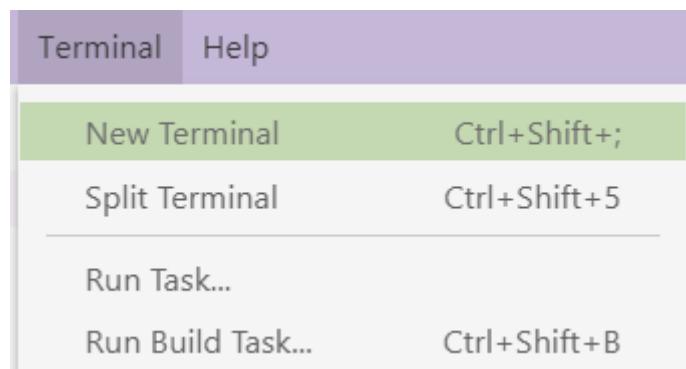
טרמינל ב-Visual Studio Code

אפשר לעבוד עם הטרמינל דרך ה-IDE החינמי Visual Studio Code, שומולץ לכל מפתח ג'אווהסקריפט. העורך הזה, מבית מיקרוסופט, הוא חינמי, לא מכבד על המחשב וגם קל להסרה. 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 ובחרו ב-
:New Terminal



ב-IDE שלכם יופיעחלון של טרמינל שנפתח כבר במיקום הפרויקט שלכם. אפשר גם ליצורתיקיות בטרמינל, למחוקתיקיות, לשנותולערורקבצים וולשות פועלות נוספות. רבים מהמתכנתים עובדים דרך הטרמינל של Visual Studio Code.

בדיקות גרסת js דרך הטרמינל

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

node -v

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

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 מקlidים `v - node` בטרמינל אתם מקבלים את הגרסה, אפשר להתקדם לשלב הבא – עובודה עם `Create React App`.

עובדה עם Create React App

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

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

לחופין, פתחו את `Visual Studio Code`, צרו באמצעותו פרויקט במיקום מסוים ובאמצעות לחיצה על לשונית `New Terminal` ו- `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 מקlid בטרמינל:

`npx create-react-app my-app`

אחר מכן פקודת הפעלה של `Node.js`.

`create-react-app` הוא שם התוכנה שאנו מפעילים, `app-my` הוא שם האפליקציה שלנו. כנספה תוכנה אמיתית אנו נקרא לה מן הסתם בשםמשמעותי יותר. כרגע נבחר ב-`app-my` – האפליקציה שלי.

אם יש לכם `Node.js` במערכת והכל תקין, ההתקנה תעבור בקלות. היא נמשכת כמה דקות טובות, אין מה לחושש. במהלך הדקוט האלו התוכנה מורידה את כל המרכיבים מהרשף על מנת ליצור אתכם במחשב סביבת עבודה מלאה של ריאקט.

```

Creating a new React app in C:\Users\barzik\local\my-app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts...

> core-js@2.6.9 postinstall C:\Users\barzik\local\my-app\node_modules\babel-runtime\node_modules\core-js
> node scripts/postinstall || echo "ignore"

> core-js@3.1.4 postinstall C:\Users\barzik\local\my-app\node_modules\core-js
> node scripts/postinstall || echo "ignore"

+ react-dom@16.9.0
+ react@16.9.0
+ react-scripts@3.1.1
added 1455 packages from 685 contributors and audited 903603 packages in 178.887s
found 0 vulnerabilities

Initialized a git repository.

Success! Created my-app at C:\Users\barzik\local\my-app
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd my-app
  npm start

Happy hacking!

C:\Users\barzik\local>cd my-app
C:\Users\barzik\local\my-app>

```

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

`cd my-app`

והפעילו את המערכת החדשה שלכם באמצעות:

`npm start`

מדובר בפקודה של Node.js שפעילה סקרייפט שנקרא `start`. אם הכל תקין, הדבר הראשון שתשים לב אליו הוא שנפתח לכם חלון של דפדפן שמכיל את האפליקציה שלכם! זה לא קסם. מה שתהileyך Node עושה הוא ליצור שרת על המחשב שלכם מאחורי הקלעים, לטען את רכיבי הריאקט וatz לפתח דפדפן שמכoon אל השרת המקומי שלכם יש מאין. אם תסתכלו על כתובות הדפדפן תוכלם לראות שמדובר בכתובת של localhost:3000. הכתובת זו מוחולקת לשני חלקים. הוא בעצם הכתובת של המחשב שלכם ו-3000 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 מצוינת.

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

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

אפליקציית הריאקט גם היא כבר מוגדרת ב-`index.js` וכוללת הפניה לקומפוננטת ריאקט פשוטה:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can
// change
// unregister() to register() below. Note this comes with some
// pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

איזו קומפוננטה? App. איך מטבחת ההפניה? באמצעות import, שנרחיב עליו בהמשך. באיזה קובץ נמצאת App? רואים את זה ב-import. בקובץ App.js:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

אם נשנה משהו בקומפוננטה, למשל נוסיף שורה כמו:

```
<p>
שלום עולם
</p>
```

ונשמר, נוכל לראות שבמיטה קסם, גם האפליקציה שלנו השתנתה ומופיע בה "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 js/node יכול ליצור צהה דבר. אבל אנחנו לא צריכים להתאמץ – Create React App יכולה לעשות את זה עבורנו. ממש נפלא. בסופו של התהליך יש לנו אפליקציה בסיסית שעבדת ואנו יכולים ליצור לה קומפוננטה ראשונה.

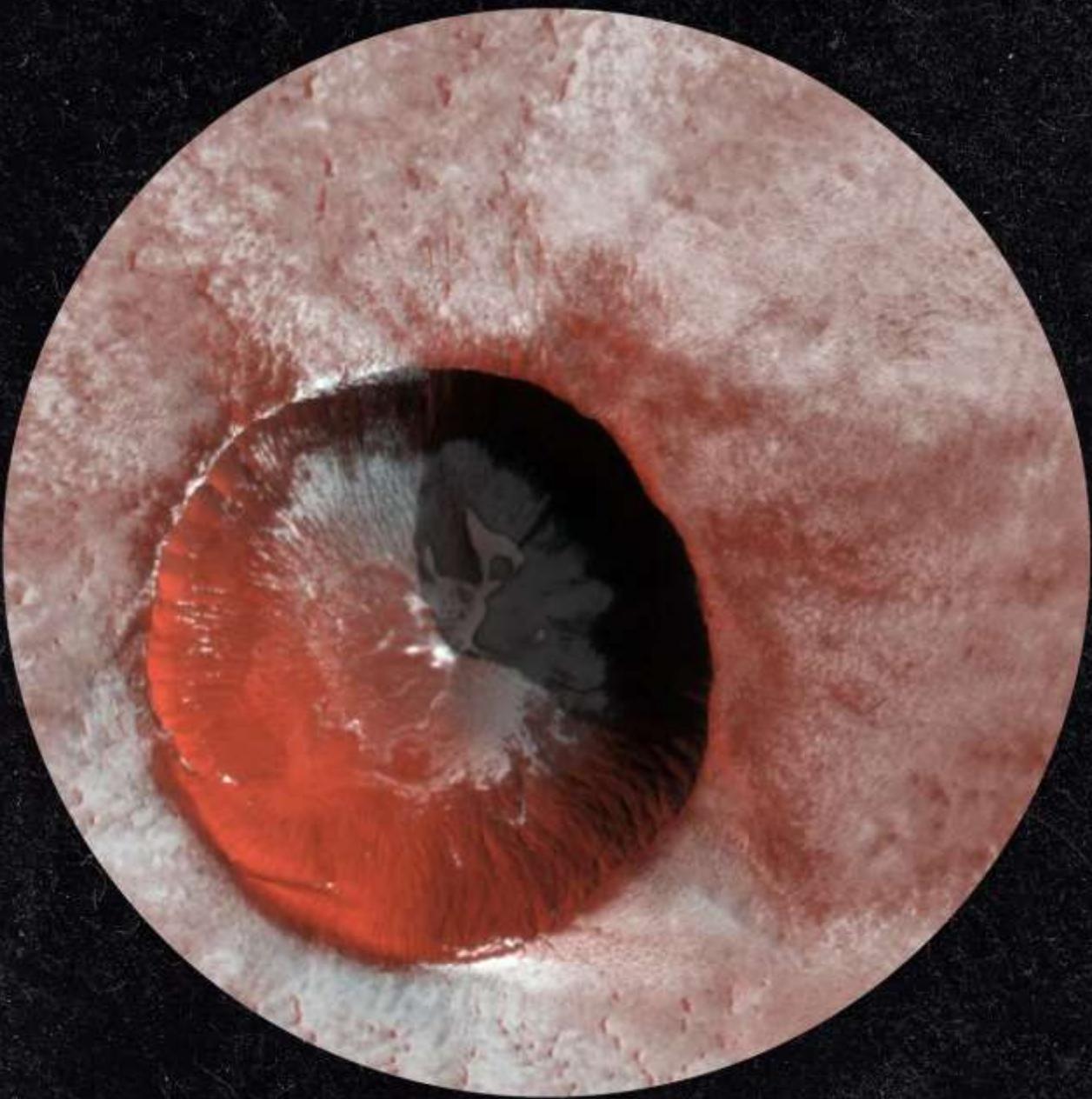
קשיים ותקלות

אם נתקלتم בבעיות בתחילת ההתקנה של Create React App, אל דאגה! יש לעיתים בעיות שנובעות משינויים במערכות הפעלה, אנטי-וירוס שפותאות מותערב או בעיה אחרת שנובעת מכש录用ות הפעלה שלנו שונות. אבל כדאי לזכור. אם יש הودעת שגיאה – פשוט חפשו אותה בגוגל. Create React App היא כל כך פופולרית, עד שיש תיעוד רב מאוד לגבייה בראשת וגמ תיעוד

של כל תקלה אחרת. קיבלתם שגיאה אדומה ומפחידה? חפשו אותה בגוגל, התיעצו לגבייה בקבוצת פיבסוק או באתר stackoverflow. אל תלגו על בניית סביבת העבודה זו כי היא הבסיס העיקרי לעובדה עם ריאקט.

פרק 4

כתיבת קומפוננטה ו Asheona CREATE REACT APP-1



כתיבת קומפוננטה ראשונה ב-React App Create

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 js.App. כל מצב חשוב מאוד: כל קומפוננטה נמצאת בקובץ נפרד משלה וועדת ברשות עצמה. מקובל מאוד שם הקובץ יהיה זהה לשם הקומפוננטה ויתחיל באות גדולה. במערכות גדולות מקובל להציב כל קומפוננטה בתיקייה משלה.

הבה ניצור קומפוננטה בשם Greeting שבה יש כתוב "שלום עולם". ראשית – הקובץ: ניצור Greeting.js תחת תיקיית src. שמו לבי שם הקובץ מתחילה ב-G גדול. הקומפוננטה זו אינה שונה שלמדו לעבוד אותה בסביבת העבודה הקודמת. היא קומפוננטה פונקציונלית שנראית כך:

```
function Greeting() {
  return <span><span>שלום עולם</span></span>
}
```

כיוון שנקרה לקובץ זהה באמצעות import, אנו צריכים ליצא את הפונקציה בדרך מסוימת. איך מיצאים? באמצעות סינטקס שנקרה export default מה פונקציה 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 חווים לעשות export בצד השני. את ה-export עושים כך:

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

אבל אם נשים רק זה, נראה שהוא מקבלים שגיאה:

'React' must be in scope when using JSX react/react-scope השגיאה זו מرمצת על כך שהוא חסר. מה חסר? במקרה הזה הריאקט עצמו. בעוד בסביבת הלימוד הפוטה שעלייה למדנו, ריאקט נמצא בכל מקום כי הכל היה בקובץ אחד, וכך אנחנו חווים לכלול בכל קומפוננטה וקומפוננטה את כל הרכיבים שאנו צריכים להפעלה. במקרה זה: ריאקט. אנו חווים "לייבא" את ריאקט ו עושים את זה באופן הבא:

```
import React from 'react';
```

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

از יש לנו כאן יבוא של ריאקט, שהוא באמצעות ויחני בכל קומפוננטה, יש לנו ייצוא, שייהיה חשוב כשרצה לשתף בקומפוננטה, ויש לנו את הקומפוננטה שלנו, שבמקרה זה לא עשו המון אלא רק מחרירה טקסט. והכל נמצא בקובץ בוודד אחד שנקרא `Greeting.js`.

הבה נשתמש ב-`App.js`. איך? פשוט נייבא אותו ונשתמש בו כרגע, כפי שלמדנו בפרק על קומפוננטה בסביבה פשוטה.

בקובץ `App.js` נעשה `import` לקומפוננטה שלנו באמצעות:

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

מהרגע שעשינו `import`, אנחנו יכולים להשתמש ב-`Greeting` איפה שאנו רוצים. למשל:

```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <Greeting />
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
```

או להשתמש בה כמה פעמים. למשל:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Greeting from './Greeting';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <Greeting />
        <Greeting />
        <Greeting />
        <Greeting />
        <Greeting />
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

זה בדיקן כמו הדוגמה פשוטה. ברגע שיש לי קומפוננטה אחת, אני יכול להשתמש בה כמה פעמים. זה שהוא בטור קובץ שאנו חנו משתמשים ב-`import export` מסתורי לא אמר להפריע לנו. בהמשך נלמד עוד על `import export`.

מהרגע שנשмарו גם את `App.js` וgam את `Greeting.js` – נראה מיד את התוצאה על האפליקציה בלי שהיא צריכה ללחוץ על F5 ולרפרש:



הקפיצה מסביבה למודית פשוטה לסייע פיתוח מרכיבת נראית קשה ומפחידה, אבל ככל שתתאמנו יותר –vr תתרגלו אליה. יש בה המון דברים טובים – ראשית יש לנו את babel שmagiu בחינם, יש לנו שרת פיתוח, בדיקות (בהמשך נלמד על הבדיקות) וגם בודק תקינות קוד של eslint, שמתՐיע על בעיות שימנו מהאפליקציה שלנו לרווח. והכל באמת בא מהקופסה. אז גם אם זה נראה מורכב, נא לא להיבחן וلتרגל על הסביבה הזאת. זו הסביבה שבסוףן של יום תפגשו במקומות העבודה.

תרגילים:

באפליקציית Create React App צרו קומפוננטה שנקראת Welcome. הקומפוננטה יושבת ב-.src/Welcome.js

בקומפוננטת Welcome יש שתי קומפוננטות נוספות: הראשונה נקראת So Iigo ומחזירה JSX שהוא Prepare to die. והשנייה נקראת Greeting ומחזירה JSX שהוא Hello, my name is Inigo Montoya. ב العمود הראשי של Create React App אני אראה שכותוב:

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.js קוראת לשתי קומפוננטות נוספות: Inigo.js ו-Greeting.js. אנו ניצור שלושה קבצים בתיקיית src: Welcome.js, Inigo.js ו-Greeting.js.

הקומפוננטות Inigo.js ו-Greeting.js הן קומפוננטות פשוטות. הדבר היחיד שאנו צריכים לזכור זה את import React from 'react'; מה שמייד מזכיר לנו export default function Inigo() { return Hello, my name is Inigo Montoya }.

```
import React from 'react';

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

Greeting.js

```
import React from 'react';

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

איך אני משתמש בהן? מבצע להן import מתוכה `Welcome.js` ומשתמש בהן ב-`JSX` כרגע. בדיקן
כפי שהשתמשתי בסביבת הפיתוח הפשוטה. כל מה שאני צריך לזכור הוא לבצע import. זה הכל.
כך `Welcome.js` תיראה:

```
import React from 'react';
import Greeting from './Greeting.js';
import Inigo from './Inigo.js';

export default function Welcome() {
  return <span><Inigo />, <Greeting /></span>
}
```

שיםו לב שאני מבצע `export default` לקומפוננטת `Welcome`.
נותר לי רק ליבא את הקומפוננטה `Welcome` למקום שאני רוצה להשתמש בה, ב-`App.js`:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Welcome from './Welcome.js';

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

export default App;
```

השווי בין סביבת העבודה הפשוטה יותר לזו של `Create React App` הוא ה-`import\export`. בעצם כל קומפוננטה עשויה `import` לדברים שהיא צריכה ו-`export` לעצמה, כדי שאחרות יוכל להשתמש בה.

פרק 5

EXPORT / IMPORT



Import\Export

מדובר בחלק אינהרנטי מג'אוועסקריפט, שחשיבותו לכל סקריפט בג'אוועסקריפט 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 כמוובן קשה מאוד לניהול ולבוגה. כפי שראינו בסביבת הפיתוח המודרנית, כדאי לעבוד עם חלוקה של קבצים עם `import` ו-`export`.

ההיגיון הוא שכל רכיב תוכנה – משתנה, פונקציה, קלאס, קבוע – יכול להיות מיובא על ידי כל רכיב תוכנה אחר. התנאי היחיד שהוא צרכיון הוא `לייצא` באמצעות המילה השמורה `export`. יכולות להיות כמה הוצאות `export` בקובץ אחד. כדי לו רכיב תוכנה שמייצא את עצמו – אפשר ליבא אותו.

במקום שיש ייבוא חייב להיות ייזוא. הייזוא הבסיסי, הפשט והקל להבנה מתקיים עם שתי מיללים שמורות: `default` ו-`export`. המשמעות של `export` היא ייזוא והמשמעות של `default` היא בירית מיוחד. כאשרנו כותבים שם של קלאס, פונקציה או משתנה ולפניהם מציבים `default export`, אנו מייצאים את אותם קלאס, פונקציה או משתנה.

הבה נדגים ייזוא פשוט. ניצור קובץ בשם `jsVars.js` שיכיל משתנים. ניצור בפרויקט של App קובץ זהה ושם נגדיר משתנה וניצא אותו. בתיקיית `src` ניצור קובץ `jsVars.js` וכל מה שייהה בו זה:

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

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

```
import React from 'react';

import logo from './logo.svg';

import './App.css';

import myVar from './Vars.js';

function App() {

  console.log(myVar);

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

export default App;
```

מה שמעניין אותנו הוא ה-`import` שמופיע בשורה הרביעית. ה-`import` מביא לתוך 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 לשכח שעם כל הכבוד לקומפוננטות, הן כתובות בג'אוועסקריפט וمتנהגות כמו כל קוד אחר של ג'אוועסקריפט. אם אני אשים `log` בפונקציה של הקומפוננטה אני אוכל לראות את המשתנה בקונסולה של הדף. אם אני אפתח את כל המפתחים, אני אקן לראות את הטקסט שעשיתי לו `export`:

```

� Vars.js  ×
src > � Vars.js
1  export default 'Hello';

... � App.js  ×
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 ליצא הכל. למשל, משתנה שיש בו טקסט או מספר:

```
const someVar = 'Hello';

export default someVar;
```

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

אנחנו יכולים לעשות `export` לפונקציה. למשל:

```
function foo() {
  return 'Hello!';
}

export default foo;
```

כדי לשים לב שמדובר בפונקציה רגילה ולא בפונקציית קומפוננטה (רואים את זה כי היא לא מתחילה באות גדולה ולא מחריצה JSX). כשאני מיבא אותה (שוב, באיזה שם שאני רוצה) אני צריך להפעיל אותה בדרך כללשה'. בדוגמה זו אני מבצע "יבוא בשורה הרובעת ומשתמש בפונקציה זו":

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import myVar from './Vars.js';

function App() {
  console.log(myVar());
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
      </header>
    </div>
  );
}

export default App;
```

אם זה נראה לכם מוכר, זו בדיק הדרך ליבא וליצא פונקציה שהיא קומפוננטה בריאקט. קומפוננטה בריאקט היא בסופו של דבר פונקציה ג'אוועסקריפטית לכל דבר, רק כזו משתמשת ב-XJS. אנחנו מיבאים אותה ומשתמשים בה ב-XJS (בניגוד לפונקציה רגילה, שלא אנו קוראים). זה הכל.

יבוא מנטייבים אחרים

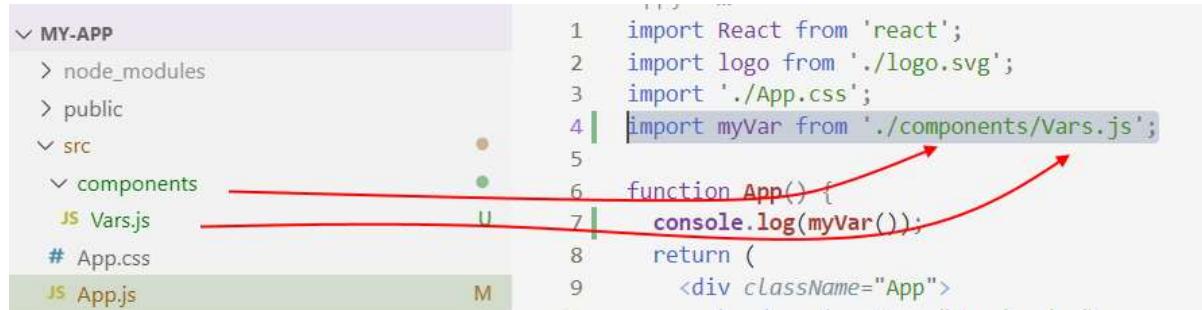
עד עכשיו ייבאו קוד שנמצא בדיק באותה תיקייה של הקומפוננטה שבה השתמשנו בקוד המייבא. אבל מה קורה אם הקוד שלנו נמצא בתיקייה אחרת? במקרה זהה אנו צריכים להקליד את הנטייב היחסי של התיקייה שלנו ביחס לקובץ שבו אנו נמצאים.

אם למשל הקוד של `myVars.js` נמצא בתת-תיקייה בשם `components`, אני איבא אותו כך:

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

ממה הכתובת זו מורכבת?

החלק הראשון - / . - מסמן את המיקום של התיקייה שלנו, הקובץ שבו אני כותב את הקוד. מהນקודה הזאת הכל מתחילה כיון שהוא צריכים לכתוב נתיב ייחודי מהמקום שבו אנו נמצאים.
 החלק השני – תת-התיקייה components.
 החלק השלישי – שם הקובץ כרגע. אם מדובר בסויימת של sz היה לא הכרחי:



The screenshot shows a file tree for a project named 'MY-APP'. The 'src' directory contains 'components' (which has 'Vars.js'), 'App.css', and 'App.js'. The 'App.js' file is open in a code editor, showing the following code:

```

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">

```

Red arrows point from the 'components' folder in the file tree to the 'import' statement in the code, and another arrow points from the 'Vars.js' file in the tree to the 'myVar' variable in the code.

כמובן, אם יש לנו תיקיות אחדנו יכולים לבצע import عمוק ככל שנרצה. למשל:

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

הענינים מסתובבים כאשר התיקייה שאנו רוצים ליבא ממנה היא לא תת-תיקייה של התיקייה הנוכחית שאנו נמצאים בה, אלא תיקייה שנמצאת מעל או תיקייה אחות. במקרה זהו אנו נעזרים ב-../ שואומר "עלות תיקייה אחות למעלה".

למשל, הבה נניח שיש לאפליקציה שלנו מבנה תיקיות כזה:

```
src/
  └── components/
    └── MyVar.js
  └── application/
    └── App.js
```

אני רוצה ש-`App` תיבא מ-`js`.`MyVar`. איך אני עושה את זה? אני צריך "עלות" תיקיה אחת למעלה `application` אל `src` ואז לרדת אל `תיקייה components`. את המסע הזה אני מתאר באמצעות `import` כר:

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

הבה נראה את המסע:
 חלק א' – / . אנו מתחילה מהתיקום שלנו.
 חלק ב' – ./ .. עולים תיקיה אחת ל-`src`.
 חלק ג' – יורדים אל `תיקייה components`.
 חלק ד' – קובץ `MyVar.js`.

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

אין צורך להשתמש בסיומת הקובץ `js`

כברית מחדל, אין צורך להשתמש בסיומת `js` של הקובץ. אפשר לעשות את ה"יבוא באוף 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 זהה ניפרד לשולם `default` וניצא יישורות את מה שאנו צריכיםלייצא. למשל:

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

הפעם, כיוון שאין לנו `default`, אנו חייבים לייבא את מה שאנו רוצים בדיק בשם שבו אנו מיצאים אותו. איך? כר:

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

או:

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

אנחנו יכולים ליבא הכל אובייקט וaz לקרו למשתנים השונים כתכונות של האובייקט באמצעות יבוא של הכל – שימוש בסימן כוכבית (*) ובמילה השמורה as. כאשר משמש ב-as * import אני נדרש לציין את תכונות האובייקט שאני רוצה להביא. לשם הדוגמה במקרה שלנו:

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

והיבוא ייראה כך:

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

ייבוא של תמונות, CSS ומשאבים אחרים

אף על פי ש-import ו-export הם תקן של ג'אווה סקריפט, בפועל מי שדואג לטעינה של המשאבים ב-App.js שعليה אנו לומדים וכן בחלק מאפליקציות הוווב המודרניות הוא הספרייה וויבפאק (webpack). הספרייה הזאת היא מודול מבוסס Node.js ואחרראית על תהליך הבילד (build) של האפליקציה, תהליך שגם במסגרת מתנהלת טיעינת הקבצים.

אנו מסוגלים להשתמש ב-import גם כדי ליבא תמונות וקובצי CSS. אפשר לראות ב-`App.js` כיצד מיבאים תמונה וגם CSS:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

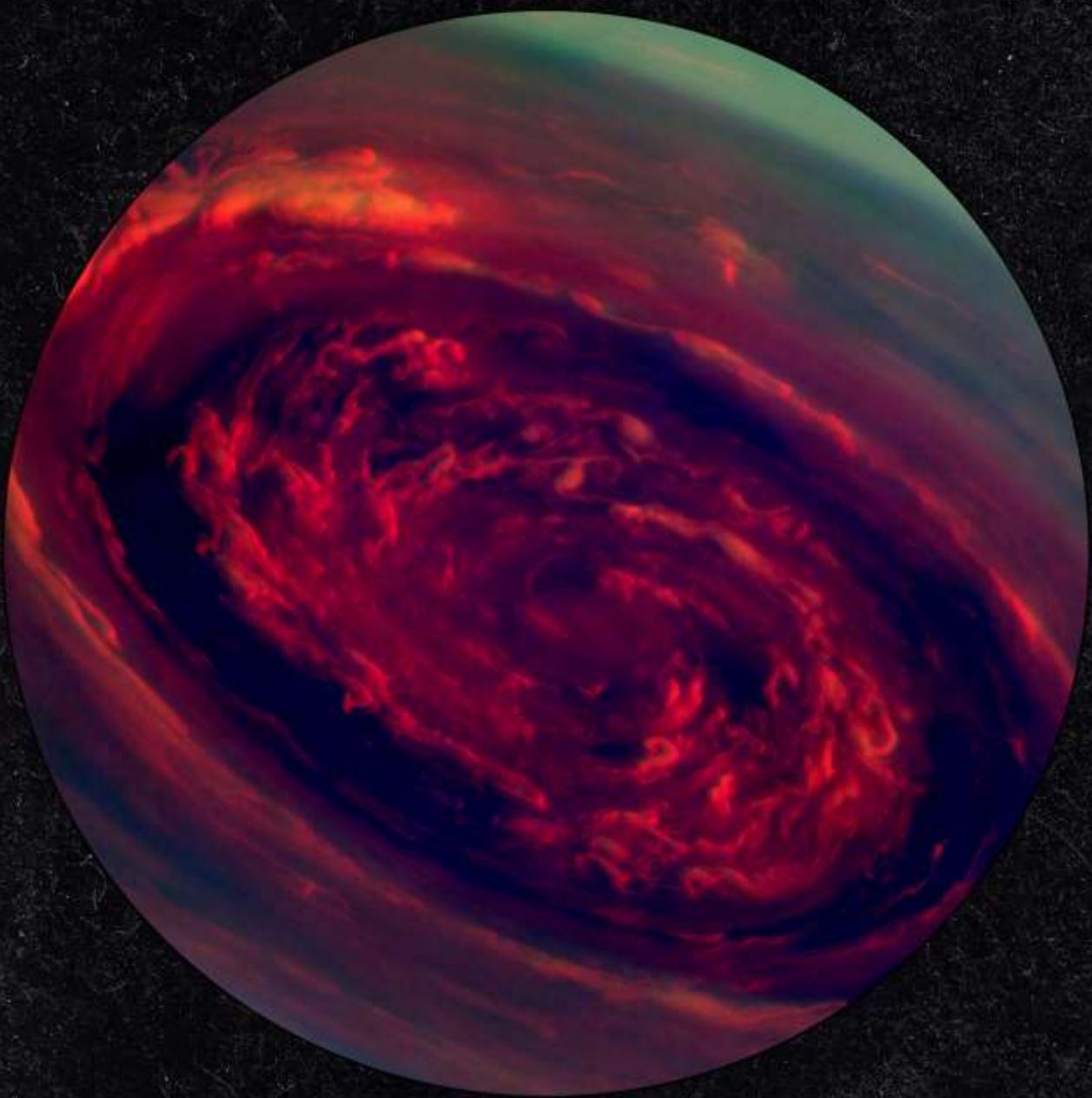
function App() {
  const someVar = true;
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
      </header>
    </div>
  );
}

export default App;
```

אני מיבא CSS וברגע שאני עושה את זה הוא מופעל ודואג לעיצוב ה-`JSX`/תמונות (ועל כך ארכיב בפרק הבא). את הייבוא והיצוא נתרגם בפרויקים אחרים.

פרק 6

JSX



JSX

עד כה לא התעמקנו כל כך ב-**JSX**, אותו HTML שנמצא בתוך ה-`return` בקומפוננטה ובכל הדוגמאות. השתמשנו "סתם" ב-HTML. אך לא מדובר ב-HTML-Ala-**JS**, סוג של XML מונחה תגיוט שמנוע עם ג'אווהסקרייפט. על מנת להציג נושא את `js`.**App** שמכיל את הקומפוננטה הראשית. כרגע יש שם HTML בלבד, אך אנו נתחילה לשחק איתו.

הכלל הוא פשוט – כל ביטוי שמופיע בין סוגרים מסוללים {} נמצא ב-**JSX** בעצם נחسب לג'אווהסקרייפט ורץ כג'אווהסקרייפט והותואתה מופיעה בקומפוננטה. כלומר, אם יש לי משתנים ואני רוצה להציג אותם, אני אכניס אותם ל-**JSX** עם סוגרים מסוללים:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  const myVar = 'Hello World!';
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        {myVar}
      </header>
    </div>
  );
}

export default App;
```

למשל, בדוגמה זו אני מכירז על משתנה `myVar` ומעביר אותו ל-**JSX**. הקומפוננטה כבר תדאג להציג אותו. אם אציג את הקומפוננטה באמצעות מבט בדף (בכתובת `localhost:3000`) אני אראה את הטקסט של המשתנה.
ובן שאני יכול להציג סוגרים מסוללים כמה פעמים שאני רוצה. למשל:

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

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

function App() {
  const myVar = 'Hello World!';
  const anotherVar = 'My Name is Ran';
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>{myVar}</p>
        <p>{anotherVar} {myVar}</p>
      </header>
    </div>
  );
}

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 React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  const myVar = <span>Hello World!</span>;
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>{myVar}</p>
      </header>
    </div>
  );
}

export default App;
```

וכן, אנו גם יכולים לשים ב-JSX שנכנס לתוך משתנה משתנים בתוך סוגרים מסוללים. העניין הוא ש-JSX הוא ג'אוوهסקרייפט טהור ויש להתייחס אליו בהתאם. אפשר לעשות אותו הכל בדיק כמו בג'אוوهסקרייפט. למשל, להכניס בתוכו ביטוי:

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

אם תציבו את JSX זהה בקומponentה שלכם, תראו שהוא מופיע 2.

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

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  const myChildren = ['Omri', 'Kfir', 'Daniel', 'Michal'];
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>{myChildren.map(child => <span> {child} </span>)}</p>
      </header>
    </div>
  );
}

export default App;
```

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

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

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

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

בתוך הסוגרים האלה אנו יכולים לכתוב כל ביטוי ג'אווהסקריפט שהוא. במקרה זה אנו עושים `map` פשוט לערך שיחזר 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 שם הילד יודפס בין ה-`child`, אנו מקיפים אותו בסוגרים מסווגלים, וזה כבר מתחילה להיות מעניין יותר. אני יכול להשתמש בו במשפט תנאי. למשל:

```
import React from 'react';
```

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

function App() {
  const someVar = true;
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        { someVar === true ? <span>I am TRUE</span> : <span>I am
        FALSE</span> }
      </header>
    </div>
  );
}

export default App;
```

אם יש לי מערך של אלמנטים, אני יכול לזרוק אותם ישירות לתוך ה- JSX והוא כבר יופיע לי אוטומטית.

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  const myChildren = [<li>Omri</li>, <li>Kfir</li>, <li>Daniel</li>,
<li>Michal</li>]

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

export default App;
```

כך למשל יוצרתי מערך של אלמנטים. כדי להדפיס אותו אני לא חייב להשתמש ב-map אלא רק לשימושו ב- JSX והוא יודפס אוטומטי. זה לא יעבוד עם אובייקט, רק עם מערך.

בעתיד, כשנראה איך המשתנים האלה מגיעים מצד השרת או מקלט של המשתמש, הכל יעשה הרבה יותר מעניין, אבל כבר עכשיו זה אמר לנו לטלטל את עולמכם. עד כה אמרנו לנו לא לערबב HTML וגו' ואוהסקcript. אבל חשוב לציין שהוא אולי נראה כמו HTML, אבל זה לא HTML. JSX שונה מה- HTML כי הוא ג'אוהסקcript. כך, למשל, אם נכתב את המילה class, שהיא וlidit לחלוטן ב- HTML אך מילה שומרה בג'אוהסקcript, נקבל שגיאה.

לדוגמא, הקוד הזה:

```
function App() {
```

```

const myChildren = [<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, כיון שהamilיה `class` היא שומרה. זו הסיבה שאנו משתמשים ב-`className`. גם קוד זהה:

```

function App() {
  return (
    
  );
}

```

לא יעבוד, כי ב-XSL אנו חיבבים להכניס אלמנט סוגר. כך למשל:

```

function App() {
  return (
    
  );
}

```

ההבדל נראה קטן, אך הוא משמעותי. לא מדוברפה ב-HTML אלא ב-XSL, ג'אווהסקריפט שעוזב עם אובייקטי XML, שאלו גם האובייקטים ש-HTML עובד איתם.

רשימות ב-XSL

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

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

air ani adfios otto? b'matzuot fonkzit map o lolet forEach. m'kobel mad od l'shtamsh b'fonkzit map shuvudat ul meorechim. b'mekra shel h'dafsat reshima zo, riaakt dorashet ma'atna l'hosif lel prut b'reshima at ha'tacona key, shmeila mazha "y'chud", b'mekra ha'za - ha'mikom shel h'prut b'reshima, am n'retsa leh'zig at ha'maruk como sh'zir b'reshima ha'zo:

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

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

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

ma shachshob fe hoa ha-key sh'mikbel at ha-index - kolmer ha'mikom shel ha'mash'tana ba'iv'er.
ha'hincha ha'mobilut ha'ya shek'oraim m'kirim at fonkzit map v'at fonkzit chz.

אם הקוד הזה:

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

נראה לכם לא מובן, כדאי ללמידה שוב על פונקציית `map` ועל פונקציית `chz`, משום שהן קרייטיות להבנה ומשתמשים בהן המונן בראקט.

תרגיל:

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

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

פתרון:

ראשית ניצור את קובץ `js/TodayTime.js` בתיקיית `src` ובתוכו ניצור את הקומפוננטה שלנו:

```
import React from 'react';

function TodayTime() {
  const today = new Date().toString();
  return (
    <div>{today}</div>
  );
}

export default TodayTime;
```

הקומפוננטה הזאת 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 מייבא ומשתמש בה) כרגע איפה שאני רוצה. במקרה הזה, ב-
:App.js

```
import React from 'react';
```

```

import logo from './logo.svg';
import './App.css';
import TodayTime from './TodayTime';

function App() {
  const someVar = true;
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <TodayTime />
      </header>
    </div>
  );
}

export default App;

```

תרגיל:

הדףו, באמצעות לולאת for, את המספרים 10 עד 1 בתוך רשימת HTML. כולם כר:

```

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

```

תזכורת:

הדףו לולאת for מ-10 עד 0 עובדת כר:

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

}

פתרונות:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  const items = [];
  for (let i = 10; i > 0; i--) {
    items.push(<li key={i}>{i}</li>);
  }
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <ul>
          {items}
        </ul>
      </header>
    </div>
  );
}

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 עם . כדי לשים לב שכיוון שאנו רוצים שה-*i* חזיר את ערכו – אנו שמים אותו בסוגרים המסתולסים. 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.

תרגום:

נתון אובייקט משתמש:

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

הדפסו את ה-key וה-value שלו כ-XJS, כך שהפלט הבא יופיע:



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

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

פתרונות:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

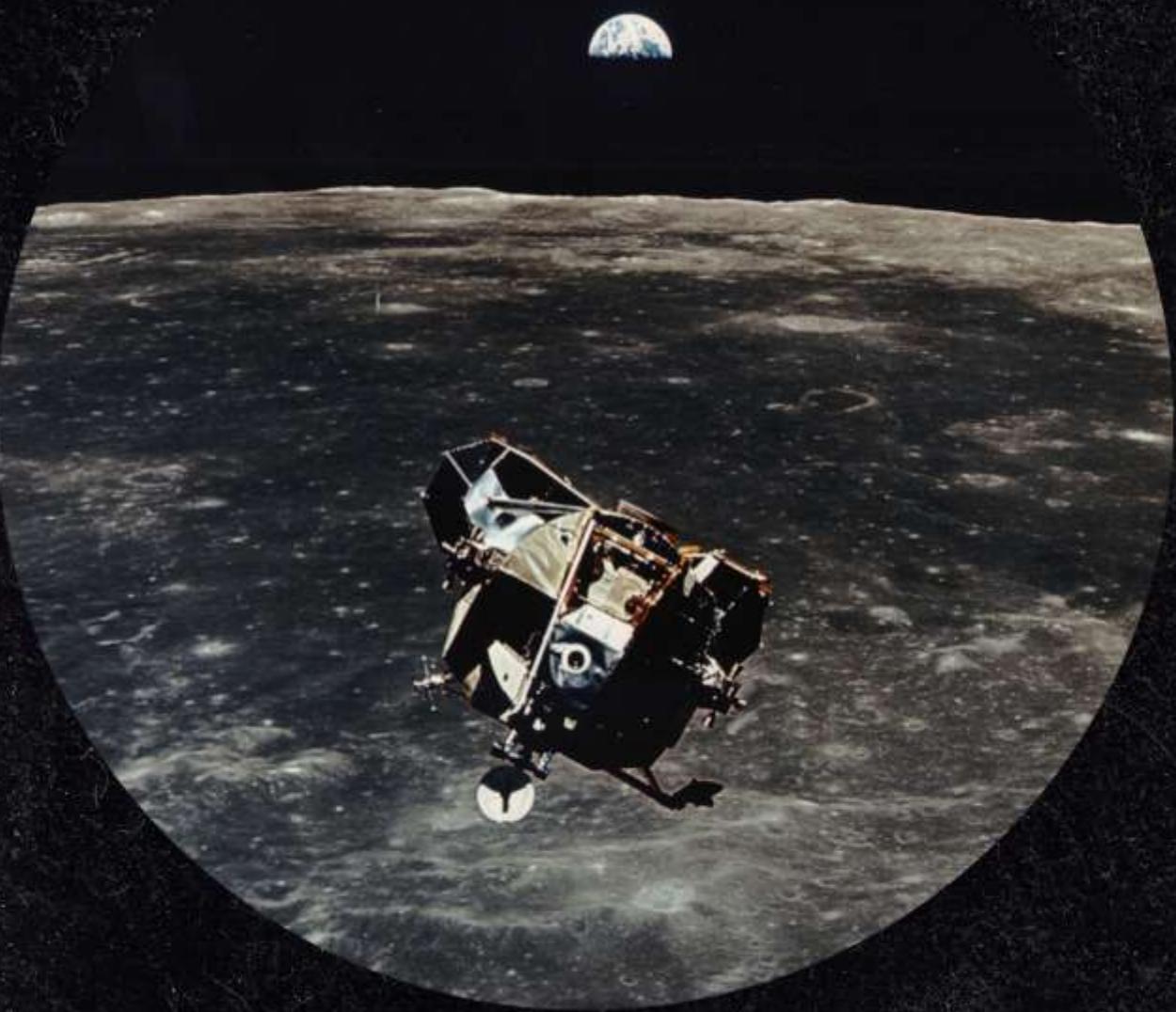
function App() {
  const user = {
    name: 'Ran',
    lastName: 'Bar-Zik',
    city: 'Petah Tiqwa',
    id: '666',
  }
  const userArray = [];
  Object.keys(user).map((value ,index) => {
    userArray.push(<span key={index}>{value} : {user[value]}</span>)
  })
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <ul>
          {userArray}
        </ul>
      </header>
    </div>
  );
}

export default App;
```

ראשית, הדקתי את האובייקט שנדרש ממוני כדי לפתח בתרגיל. שנית, יצרתי פונקציית `map` ואת `h-XJS` והכנסתי אותו לתוך מערך באמצעות `push`. השלב הבא הוא פשוט לשים את המערך זהה בתוך `h-XJS` ולסמן עלייו שירנnder 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 במקומות שיש פלט יש גם קלט – היכולת שלנו לשלוח נתונים לקומפוננטה ובהתאם לו לקבל פלט. כך, למשל, אני יכול לשלוח שם והקומפוננטה תחזיר לו: "בוקר טוב, רן" או "ערב טוב, רן" בהתאם לשם שאני שלח לה. אני יכול ליצור קומפוננטת טבלה ושלוח לה מידע בפורמט JSON כדי שהקומפוננטה תציג את המידע בפורמט טבלאי. אני יכול ליצור קומפוננטת טופס שיש בו פרטיהם וכו'. הקלט הוא קרייטי בפייתו מונחה קומפוננטה ובדרך כלל נמצא 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 `jsx.Welcome`. אני רוצה שתתשו לב שכעת, כשאנו עובדים עם קומפוננטות מסודרות, נהוג לכתוב את סימנת הקומפוננטות כ-`-c-Welcome` ולא `-c-Welcome`, אם כי שתי השיטות יעבדו.

```
import React from 'react';

function Welcome() {
  return (
    <span></span> // Component will be here
  );
}

export default Welcome;
```

עכשו נוסיף את הטקסט שלנו:

```
import React from 'react';

function Welcome() {
  const name = '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 שאולן אצטראר לשנות את הנתיב, כפי שלמדנו בפרק על import ו-export):

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

ואשתמש בה בכל מקום פה:

```
<Welcome />
```

מה הבעיה? הבעיה 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 אכנס `props` כפרמטר לפונקציה. זה יראה כך:

```
function Welcome(props)
```

ה-`props` הזה הוא בעצם אובייקט הקלט שלנו. יש בו כל הפרמטרים שנעביר לקומפוננטה. איך אנחנו מעבירים? באמצעות התכונה של הקומפוננטה, שנראית בדיק כmo תכונה של HTML. הנה, כך:

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

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

```
import React from 'react';

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

export default Welcome;
```

כלומר אנו מעבירים פרמטרים באמצעות תכונות. תכונות של תגיות 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 לתקן שום רכיבי HTML ולשנות אותם. אותו הדבר עם קומפוננטות ריאקטיות. באמצעות שימוש בתכונות אנו מעבירים להן מידע.

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 צזה דבר? בקומפוננטה שלי אני אגדיר את הדרך שבה שולחים לי את התואר, למשל prefix:

```
import React from 'react';

function Welcome(props) {
  const name = props.name;
  const prefix = 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 במשלוח מחרוזות טקסט. אני יכול לשלחו הכל. ב- JSX אני יכול לשלחו למשל אובייקטים שלמים. בואו נניח שיש לי אובייקט משתמש עם שם ותואר, שהוא צזה:

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

איך אני שולח את האובייקט הזה?

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:

```
import React from 'react';

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

export default Welcome;
```

דרך נוספת היא להשתמש ב-destructuring. מדובר בדרך מיוחדת שבה אנו לוקחים אובייקט ומפרקים אותו למערך. אני מעביר את האובייקט שלו ב-props אחד:

```
function App() {
  const user = {
    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 במשתנים השונים הuliar מבודד שנקרו destructuring. אני בעצם לוקח את האובייקט וממיר אותו למשתנים. שם המשתנה חייב להיות זהה לשם האובייקט. למשל:

```
import React from 'react';

function Welcome(props) {
  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 כזכור גם מ-`prefix` וגם מ-`name`. במקום לעשות משהו צזה:

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

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

אני עושה משהו כזה:

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

זה בדוק אותו הדבר.

כך או אחרת, זה הדבר הכי חשוב בקומפוננטה, לקבל פרמטרים פנימה. זה הקולט שלנו ואיתו אנו עובדים. הפלט? הפלט במקרה הזה הוא מה שנוצר כתוצאה מレンדר ומה שהוא רואים. בהמשך נלמד על כך שהפלט האמתי הוא האירועים שמשופעים בקומפוננטה.

תרגיל:

צרו קומפוננטה בשם Birthday מקבלת מספר באמצעות תכונת age ומציג אותה בטקסט באופן הבא:

Happy Birthday! You are X years old!

כש-X הוא הגיל. קראו לקומפוננטה זו מ-.app.js

פתרון:

הקומפוננטה נשמרה בשם `xsj.Birthday` והוא מכילה את הקוד הזה:

```
import React from 'react';

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

export default Birthday;
```

ב-.js.app היא תיקרא כך:

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

אפשר לראות שהדבר היחיד שמשמעותו CAN הוא שיש לו props ומהם אני לוקח את ה-age. כשאני קורא לקומפוננטה זהו, אני מעביר לה age, בדוק כפי שלמדנו.

תרגילים:

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 App.js מושלמת להיראות כך:

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

פתרונות:

קומפוננטת Birthday:

```
import React from 'react';

function Birthday(props) {
  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 בדוק כמו:

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

אבל השימוש ב- destructuring אלגנטי יותר.

תרגילים:

הารכיטקט הבכיר בחברה קבע שהקומפוננטה שלכם תקבל את האובייקט של ה-user, בסגנון זהה:

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

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

פתרונות:

כיוון שכך עליינו שינוי, אנו חייבים לעבוד עם האובייקט. אם באמצעות השימוש ב-destructuring, יהיה לנוprops.age ו-props.name שבמקוםprops.user יש לנו:props.user

```
import React from 'react';

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

export default Birthday;
```

תרגילים:

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

פתרונות:

אסור לנו לשכוח ש愧 על פ' שב-XSL עסקינו, עדין מדובר בג'אווסהקרייפט פשוטה וקללה, שאנו מכירים. אם אתם מתכנתים, אתם אמורים לשלוט במשפטי תנאי או במשפטי תנאי מקוצרם. הדרך להכניס לקבוע טקסט לפי הגיל היא באמצעות תנאי מקוצר:

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

או באמצעות משפט תנאי פשוט יותר אך אלגנטי יותר:

```
let phrase;

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

לא משנה באיזו דרך בחרתם, אתם יכולים להשתמש בששתנה אחד כדי ליצור משתנה אחר ולשים אותו במה שהקומפוננטה מחייבת:

```
import React from 'react';

function Birthday(props) {
  const { age, name } = props.user;
  const phrase = 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-XSL ולסינטקס המבלבל – זה בסופו של דבר ג'אווהסקריפט.

פרק 8

דיבאת



דיבאג

חלק שימושוטי מכל פיתוח תוכנה הוא הילך הדיבאגינג – כולם מציאת התקלות, השגיאות והבעיות שיש בקוד שננו מרכיבים. דיבאגינג פירושו מציאתאגים, וכשאנו כתבים קוד תמיד, אבל תמיד, יהיו בעיות.

בדומה לג'אווהסקריפט, אנחנו יכולים לבצע דיבאג לכל קומפוננטה ריאקט עם פירפוקס או כרום בקלות ובעילות שננו מרכיבים את סביבת הפיתוח שלנו. בנוסף על כך, יש לכרום ולפירפוקס توוסף של כל מפתחים ייעודי לריאקט שאוטו כדי להכיר ולהתקין עכשו. חפשו "React Developer Tools" בחנות התוספים של הדפדפן והורדו אותו או הקliquו על:

לכרום <http://bit.ly/reactdevtool>

או על:

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

[Home](#) > [Extensions](#) > React Developer Tools



React Developer Tools

Offered by: Facebook

★ ★ ★ ★ ★ 1,195

Developer Tools



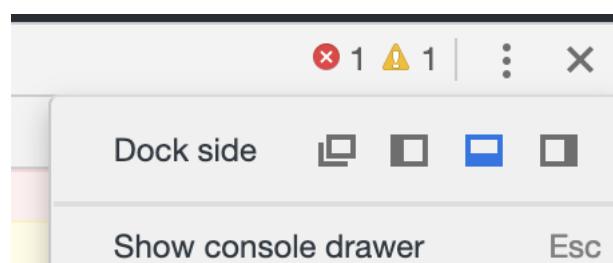
1,833,536 users

על מנת להציג את תהליך הדיבאג, נחוץ לנו קוד לדבג. אתם יכולים להשתמש בכל קוד שכתבתם עד כה. בפרק זהה אני אציגים על הקומפוננטה שיצרנו בתרגילים בפרק הקודם. אבל התהילה עובד, כמובן, בכל קומפוננטה שהיא ובכל קוד שהוא. הדיבאגר המובנה בדף והתוסף המויחד שמחלו אליו אמורים להיות ידיכם הטוביים ביותר.

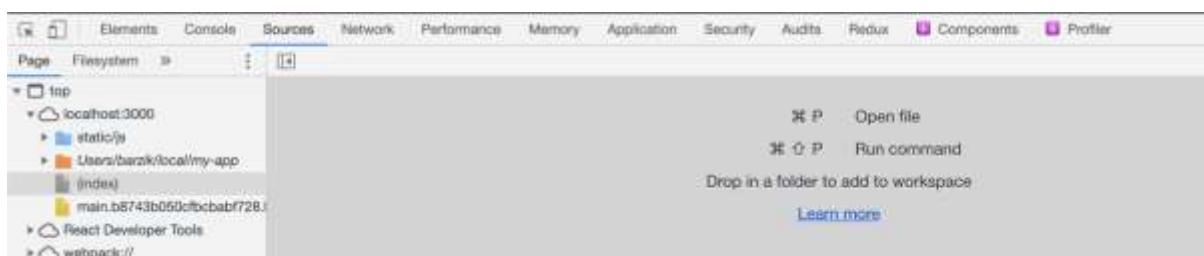
ראשית, הדיבאגר של הדפדפן עובד כרגיל. בכל דףדף מודרני יש כלי מפתחים מובנה ובפרק זה נלמד על כירום, הדפדפן הנפוץ ביותר. אך יש לנו מפתחים זהה בכל דףדף עם אותו מבנה. בכרום אנו מפתחים את כל הפתחים באמצעות צירוף המקשים קונטROL, שיפט ו-- בחלונות או בלינוקס וקומנד, אופשן ו-- במק. יפתח לכם כל הפתחים בתחתית המסך.

היכנסו אל האפליקציה של Create React App והריצו אותה באמצעות start npm. לאחר שהאתר נפתח, פתחו את כל הפתחים.

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



מעבר לtagית sources:



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

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

The screenshot shows the React Developer Tools interface with the 'Sources' tab selected. On the left, a tree view shows the project structure: 'App' > 'localhost:3000' > 'static' > 'Users/barzik/local/my-app' > '(index)' > 'main.b8743b050cfbcbabf728.js'. The right pane displays the code for 'Birthday.jsx':

```

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;

```

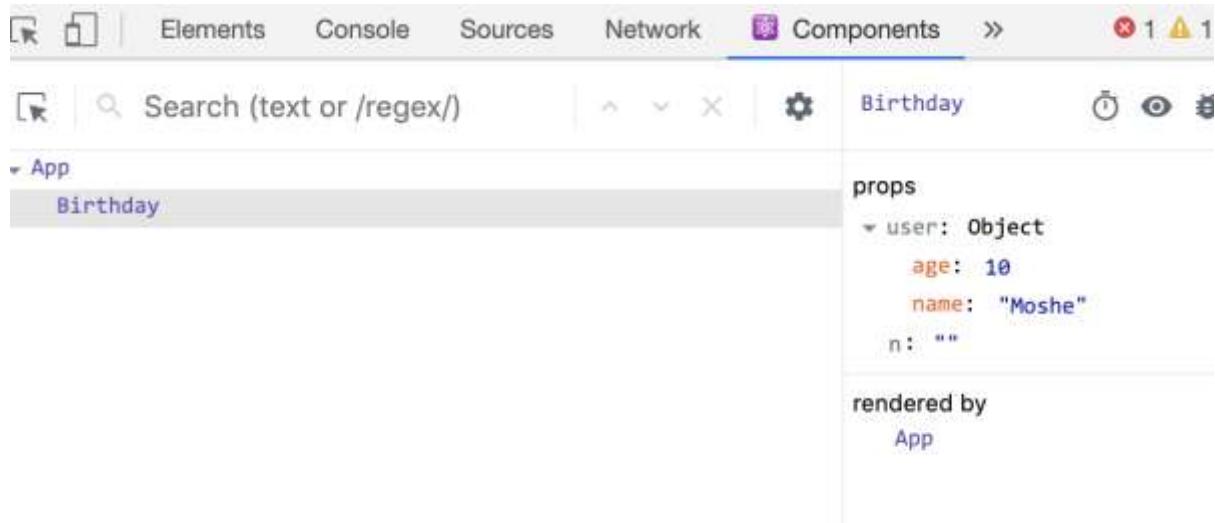
אנו יכולים, באמצעות לחיצה בצד השורה המתואימה, ליצור נקודת עצירה (Breakpoint). מדובר בנקודת שבה הסקריפט שלנו עוצר ולא ממשיך הלאה ואנו יכולים לראות מה מצב הקוד, לבדוק ולראות את ערכי המשתנים ולאחר מכן המשיך את הרצה. אם נעה מחדש את העמוד (באמצעות F5 למשל או קונטרול + z) התוכנה תעוצר בנקודת העצירה זו ונוכל לראות את ערכי המשתנים, להקליד בקונסולה ולצעוד קדימה בזיהירות כדי להבין מה השتبש:

The screenshot shows the React Developer Tools interface with the code editor open to 'App.js' and the debugger sidebar open. A breakpoint is set on line 5 of 'Birthday.jsx'. The debugger sidebar shows the following state:

- Paused on breakpoint**
- Watch**
- Call Stack**
- Scope**
 - Local**
 - age: 18
 - name: "Moshe"
 - phrase: undefined
 - Props**: {user: {...}}
 - This**: undefined
 - _props\$user**: {name: "Moshe", age: 18}
 - Closure**: ./src/Birthday.jsx
 - Global**
- Breakpoints**
 - App.js:12
 - App.js:15

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

אר התקנת נוספת הפיתוח של ריאקט פותחת לנו אפשרות נוספת. החשובה ביותר היא צפייה בראשימת כל הקומפוננטות. אם התקנתם את תוסף הפיתוח, תוכלם לראות לשונית נוספת בשם Components. לחיצה עליה תציג לכם את רשימת כל הקומפוננטות. בתוך כל קומפוננטה תוכלם לראות את מצב ה-`props` שלה ובאייזו אפליקציית ריאקט היא נמצאת:



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

להרבה מפתחים מתחילה יש נטייה להימנע מהડיבאג כיון שמדובר בכל שנראה מפחיד וגם כך יש המון מה ללמידה – אבל זה מצער. כדאי לשקיעע כמה שעות בניסיון להבין איך הוא עובד ולהשתמש בו במקום לנסות לפענוח מה הבעיה בעצמכם או בעזרתו כל מיני מעקרים כמו `console.log()`. זה הרבה יותר קל ומהיר, ובטוח/arour - משתמשים מבחןת הזמן. כל המפתחים משתמשים ומשתכללים כל הזמן, וכך גם התוסף של ריאקט. יש סיכוי שבזמן שתתקראו את הפרק, התצוגה של התוסף או של כל המפתחים תשתנה – הם עוברים שינויים כל הזמן. אבל הפקנציונליות הבסיסית של כל המפתחים לא משתנה – מקום לשיטים נקודת עצירה, להריץ קדימה ולראות מה מצב ה-`props`. אלו הדברים החשובים לכם.

מבחינת תרגול – נווטר עליו בפרק זהה. התרגול שלכם יהיה בשימוש בדייגר בפרק הבאים.

פרק 9

סיטייט



סטייט

עד כה יצא לנו לעבוד עם קומפוננטות סטטיות בלבד, ככלומר-Calmer שיש בהן רק `return` של JSX או שמקבלות `props` ועוד מחרוזות JSX. קומפוננטות אלו הן מצינות אבל "חסורות זיכרון". ככלומר, אני מעביר לקומפוננטה מידע, היא מרנדרת אותו וזהו. גם אם אני אשנה את המידע, אחרי שהקומפוננטה הוצאה, אני עדיין אראה את המידע שהכנסתי בהתחלה.

אדגים את הבעיה עם קומפוננטה שנקראת `CountUp.js`. זהה קומפוננטה שעשושה משהו פשוט מאוד - היא מציגה כפטור ובכל פעם שאני לוחץ על הכפטור, הספרה עולה ב-1.

air אני ממש דבר זהה בג'אוויסקcript? בקלי קלות עם אירוע שצמוד לכפטור ויפעל פונקציה כזו:

```
let count = 0;

function countUp() {
  count = count + 1;
}
```

לא שהוא מרכיב מדי.

אני יכול להכניס את הקוד הזה בקומפוננטה באופן הבא:

```

import React from 'react';

function CountButton() {

  let count = 0;

  function countUp() {
    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 שהגדרנו בתוך הפונקציה תהיה זמינה אך ורק בתוך הסקוב של הפונקציה.

שימוש לב: הפונקציה שאנו מפעילים ב-`onClick` מוקפת גם היא בסוגרים מסוללים כיון שאנו רוצים ששםה יתורגם לקוד. אנו נמצאים ב-XSL וצריכים להבהיר שמדובר בפונקציה של הקומפוננטה.

אם אני אקרא לקומפוננטה זו ב-`-js.jsx` באופן הבא:

```

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

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <CountButton />
      </header>
    </div>
  );
}

export default App;

```

אני רואה שכאן הקומפוננטה מתרננרט ומוצגת הספרה 0, אך הספירה לא מתקדמת בלחיצת הceptor. אם תציגו בדיינגר, אולי למדנו בפרק הקודם, תוכלו לראות שכאן הfonkציה countUp נקראת בכל לחיצה, אבל המספר לא מתעדכן. מדוע? כדי להבין את זה לעומק אנו צריכים להבין יותר מהו רנדור.

ריאקט ורנדור

כאשר אנו כותבים קומפוננטה ב-`JSX` של האפליקציה שלנו, אנחנו לא מייצרים `HTML`, אלא אובייקט ריאקטי שריאקט מכניסה ומציגה אותו ב-`DOM` האמיתי. פועלות ההציגה זו, שמתרגמת את `JSX` לתצוגת `HTML` שהדף יודע לטפל בה, נקראת "רנדור" מלשון `render` – הרצה. הרנדור הראשון של כל קומפוננטה מתבצע בטיענה הראשונית ואנו רואים את התוצאה שלו בדף. כל עוד אין רנדור חדש, התצוגה בקומפוננטה לא תשתנה. זה בדוק מה שקרה עם הקומפוננטה `CountButton`. הפונקציה שנראית מהceptor אכן עובדת וمعدכנת את המשתנה הפנימי, אבל כל עוד אין רנדור חדש, לא נראית את השינויים שנעשו בקומפוננטה אנו ח"בם לבצע רנדור מחדש.

על מנת להציג את השינויים שנעשו בקומפוננטה אנו ח"בם לבצע רנדור מחדש. הרנדור חדש נעשה באמצעות אחת משתי דרכים:

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

ברגע שיש רנדור חדש, הקומפוננטה מתעדכנת גם ה-XSL, ואנו רואים את הערכים החדשניים ב-XSL. בדוגמה שלנו, אם אנו רוצים שהקומפוננטה תציג את הערך של `z` שהשתנה, אנחנו צריכים לדאוג לשנות את הסטייט הפנימי שלה. בשביל זה אנחנו צריכים לדעת מה הוא סטיטי.

סטיטי

סטיטי הוא בעצם הדינמיות של הקומפוננטה. לקומפוננטות שעבדנו איתן עד כה לא היה זיכרון. או שהן קיבלו את המידע שלהם מבפנים (קומפוננטות תצוגה) או שהן קיבלו את המידע שלהם מקומפוננטת אב דרך `props`. כך או אחרת, הן קיבלו את הקלט, עשו איתו חישוב והעבירו אותו ל-XSL. הקומפוננטה רונדרה, ה-XSL הוצג בתור HTML ובעצם הסיפור הסתיים. פה אנו זקוקים לזכרון פנימי שיאפשר לקומפוננטה לנחל את המשתנים שלה, וחשוב מכך – לסמן לריאקט מתי לבדוק לרנדר את הקומפוננטה מחדש. הסטייט במקורה של ריאקט הוא אובייקט שאנו מכנים אליו את המידע שהוא רוצים "לזכור". כאשר אנו משנים את המידע שיש בזיכרון, יש לנו רנדור מחדש.

از איך מנהלים את הסטייט? בקומפוננטות מסווג פונקציות אנו משתמשים במנגן פשטוט שנקראו הוקים (hooks) או ביחיד הוק. בהוק אנו מבצעים שתי פעולות:

1. הגדרת החלק שהוא מכניסים לסטיטי.
2. הגדרת הפונקציה שבאמצעותה אנו משנים את החלק הזה בסטייט. בקריאה לפונקציה זו מתבצע הרנדור מחדש.

זה נעשה באמצעות הhook `useState`. הוק הוא שם מפחיד לפונקציה פשוטה שמקבלת פרמטר אחד ייחיד – הערך ההתחלתי של הסטייט שלו. הפונקציה מחזירה מעורר עם שני חלקים. החלק הראשון הוא הסטייט עצמו, שבו אני יכול להשתמש ב-XSL (או בכלל מקום בקומפוננטה), והוא לקרוא בלבד. החלק השני הוא הפונקציה שבאמצעותה אני משנה את הסטייט.

ראשית נראה וזה נסבר. 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 React, { useState } from 'react';

function CountButton() {

  const [count, setCount] = useState(0);

  function countUp() {
    setCount(count + 1);
  }

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

export default CountButton;

```

ראשית, אני קורא להוקים באמצעות import. ה-import זה משתמש ב- destructuring של ממדנו עליי בפרקם קודמים. קיבלנו אותו כמות שהוא - בכלל 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 המציג את ערך הסטיטו, האיבר השני הוא הפונקציה שמשנה אותו. מקובל (אך לא חובה) לכתוב את שם הפונקציה כך שייתחיל ב-set ואז יבוא שם הסטיטו באות גדולה.

למשל, אם המשתנה שרציתי להכניס לזיכרון הוא count, שם הפונקציה שמשנה את המשתנה בזיכרון של הקומponentה יהיה `count`. במקרה שלנו שם המשתנה הוא: count, אך שם הפונקציה שמשנה את המשתנה בזיכרון יהיה `setCount`. זה הכל.
אני מבצע את ההגדרה זו באמצעות השורה:

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

הפונקציה המובנית useState היא הפונקציה הקרייטיתפה, כיוון שהיא מקבלת את הערך הראשון של המשתנה שלי, במקרה זה 0.

עכשו אני צריך לשנות בקומפוננטה ולהמיר את כל הפעולות שבן אני משנה את הערך של `count` בפונקציה של `setCount`, הפונקציה שאומרת בעצם – תכניס לזכרון את הערך הזה, במקרה שלנו – `count + 1`:

```
setCount(count + 1);
```

זה הכל!

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

שם הפעולה	הקוד החדש	הקוד הישן
להביא את ההוקים	<code>import React, { useState } from 'react';</code>	<code>import React from 'react';</code>
לקבוע ערך ראשון לסת依ט ולקבל את המשתנה לזכרון, לקבל פונקציה שמשנה אותו	<code>const [count, setCount] = useState(0);</code>	<code>let count = 0;</code>
לשנות את הערך	<code>setCount(count + 1);</code>	<code>count = count + 1;</code>

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

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

```

import React from 'react';
function ShowHover() {
  let time = 0;
  function countHover() {
    time = time + 1;
  }
  return (
    <div>
      <h2 onMouseOver={countHover}>{time}</h2>
    </div>
  );
}
export default ShowHover;

```

אם תציבו את הקוד זהה ב-`index.js` ותקרוו לקומפוננטה מתוך `index.js`, תראו שלא משנה כמה פעמים אתם עוברים על הcptor, המספר 0 ישאר על הלוח. למה? כי הקומפוננטה לא עברה רנדור מחדש. נכון, הקליק שינה את המשתנה הפנימי, אבל כל עוד אין רנדור מחדש, אנו רואים את תוצאה הרנדור הקודם גם אם ה-`time` משתנה.

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

1. לבצע `import {useState} from 'react'`, במקורה הזה `useState`.
2. לומר לקומפוננטה בהתחלה איזה משתנה נכנס לזיכרון (במקורה שלנו `time`) ולהגדיר פונקציה שתשנה את המשתנה בזיכרון. שם הפונקציה הוא תמיד `set` ואז שם המשתנה מתחילה באות **גדולה**.
3. לשנות את הקומפוננטה שצריך באמצעות הפונקציה שמשנה את המשתנה.

```
import React, { useState } from 'react';
function ShowHover() {
  const [time, setTime] = useState(0);
  function countHobver() {
    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 בלבד, אולם הם חלק חשוב מאוד בראקט
ואנו נלמד בהמשך על דרכי管理 סטייט גלובלי. בנוסף על כך, נלמד גם על עוד הוקים.

תרגילים:

צרו קומפוננטה בשם `CountDown` שיש בה כפטור ומספר שמתחל ב-10. הקומפוננטה תספור מהמספר זהה לאחר (אין צורך לעזר ב-0).

פתרונות:

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

הקומפוננטה זו עשוה לבדוק 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 שלושת החלקים:

1. ה-`import` השונה (מייבאים את הhook של `useState`).
2. קובעים את המשתנה שיכנס לזכרן של הסטיט ושם הפונקציה שמכניסה את המשתנה לזכרון. שם הפונקציה תמיד מורכב מ-`set` ושם המשתנה. ניתן לקבוע את הערך ההתחלתי של הסטיט בשלב זהה.
3. כשרוצים לשנות את המשתנה, משתמשים תמיד בשם הפונקציה שמכניסה את המשתנה לזכרון.

תרגילים:

בקומפוננטה הקדמת שיצרתם, צרו קומפוננטה שמקבלת את המספר ההתחלתי ב-timeprops בשם time. וכן הספירה תיפסוק ב-0.

פתרונות:

```
import React, { useState } from 'react';
function CountDown(props) {
  const [count, setCount] = useState(props.time);
  function countUp() {
    if (count > 0) {
      setCount(count - 1);
    }
  }
  return (
    <div>
      <h2>{count}</h2>
      <button onClick={countUp}>Clck me</button>
    </div>
  );
}
export default CountDown;
```

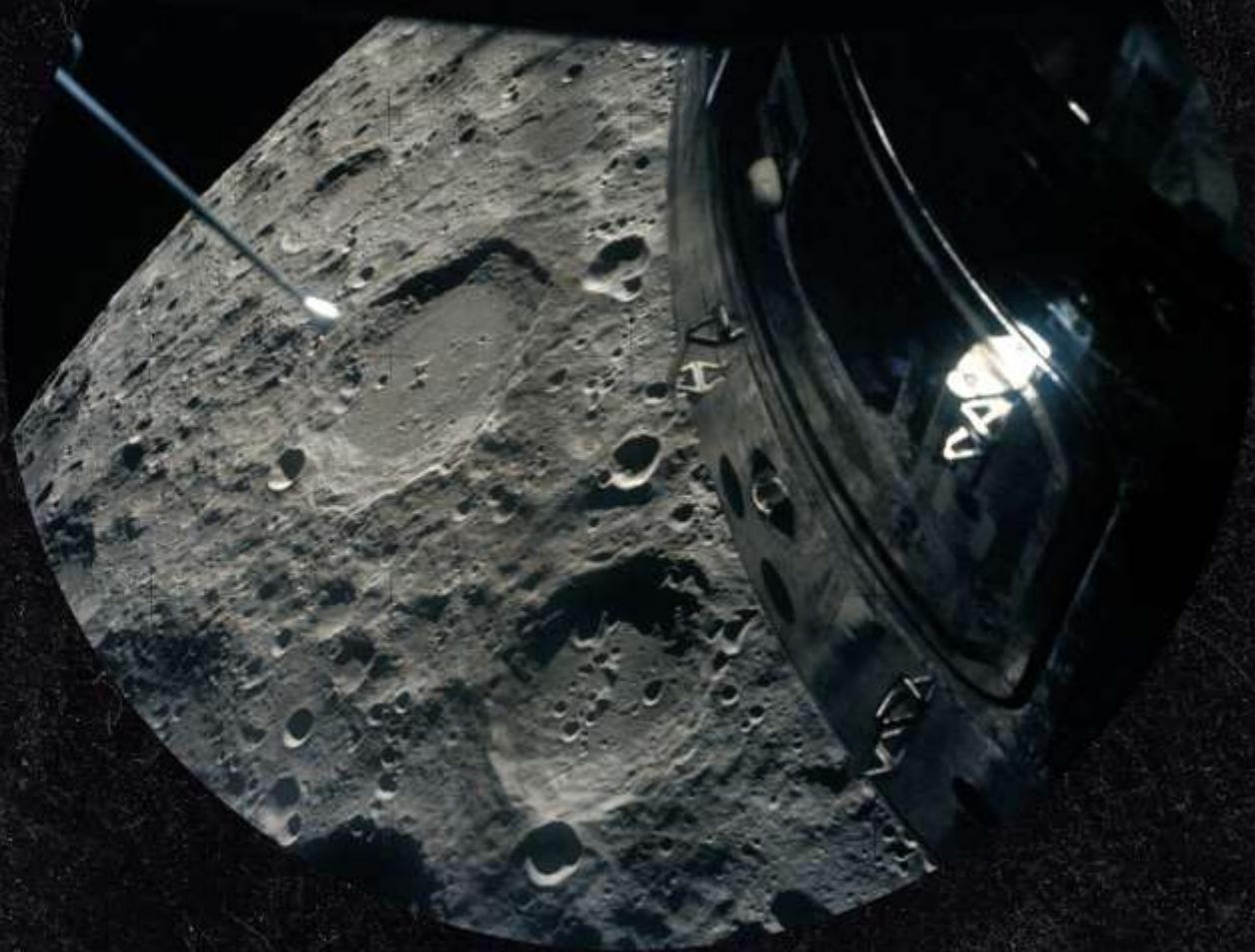
אני קורא لكומפוננטה זו כМОון כך:

<CountDown time="3" />

הפתרון כמעט זהה לפתרון הקודם. המטרה היא להבין שהסטיטיים אינם תורה מס'ני. הם פשוט משוחה ששיםם בזיכרון, זה הכל, ואם לא משנים להם את הערך אפשר לגשת אליהם כרגיל.

פרק 10

תכנון מבנה הكومפוננטות



תיכון 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 אחרות, ואפשר לחת את הקומפוננטות ולשים אותן בכל מקום אפשרי.vr כר למשל אם יש לנו קומפוננטה המציגת טקסט בצורה נאה או מיוחדת, ניתן להשתמש בה בהמון קומפוננטות אחרות. לדוגמה, קומפוננטה המציגת טקסט לא ממש אכפת מה היא מציגת כל עוד היא מקבלת את המידע `props` המתאים.

הבה נניח שיש לנו קומפוננטה שהקלט שלה (כלומר מה שהיא מקבלת ב-`props`) הוא מילישניות והפלט שלה הוא זמן מעוצב בצורה נאה וקראית. שם הקומפוננטה הוא `xsj.js`:

```
import React from 'react';

function Watch(props) {
  const date = new Date(props.milliseconds)

  const options = { weekday: 'long', hour: 'numeric', minute: 'numeric', second: 'numeric' };

  const time = 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 אזכיר קומפוננטה אחרת שמעבירה אליה את הזמן שאנו רוצה שקומפוננטת הבית תציג. למשל, קומפוננטת `xsj.js` `TodayTime` שמעבירה את הזמן של היום אל קומפוננטת `xsj.js` `Watch`:

```
import React from 'react';
```

```

import Watch from './Watch.jsx';

function TodayTime() {
  const today = Date.now();
  return (
    <Watch milliseconds={today} />
  );
}

export default TodayTime;

```

את קומponentת `jsx` אני שם בכל מקום שבו אני צריך את השעה הנוכחית. אבל אם אני אציב את הקומponentה זו במקום כלשהו – למשל `js`-`app`, הדף הראשי של אפליקציית הריאקט – אני אראה שהוא ממש מרגץ:

```

import React from 'react';
import './App.css';
import TodayTime from './TodayTime';

function App() {
  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 – גם תרגלו – אתם כבר יודעים שהרנדור של הקומponentה רץ פעמי אחד בלבד וועל מנת לעדכן אותו אנחנו צריכים לעשות משהו – כמו למשל `setInterval`. מה הבעיה? אני יכול לשים את ה-`val` `setInterval` בתוך הקומponentה, אבל אני לא רוצה לשים אותו ב-`Watch.jsx`. מדוע? לערבות לוגיקה עם קומponentת תצוגה זה לא רעיון טוב.

בתחילת הפרק כתבתי שחלק מתכוננו נכון הוא תכונן קומפוננטות קטנות כל האפשר. אבל מעבר לתיאוריה – אם אני אציב `setInterval` ב-`Watch`, אני לא אוכל להשתמש בה במקומות אחרים – כמו למשל במצב תצוגה של שעון קבוע (לדוגמה: השינוי האחרון באתר בוצע בשעה כך וכך). מי שצריכה להעלות את הזמן בכל שנייה ולהציג את הזמן בסטייט שלה היא קומפוננטת `TodayTime.jsx`.

את זה ניתן לעשות בקלות עם `setInterval`:

```
import React, { useState } from 'react';
import Watch from './Watch.jsx';

function TodayTime() {
  const [today, setToday] = useState(Date.now());

  function upTime() {
    setToday(Date.now());
  }

  setInterval(upTime, 1000);

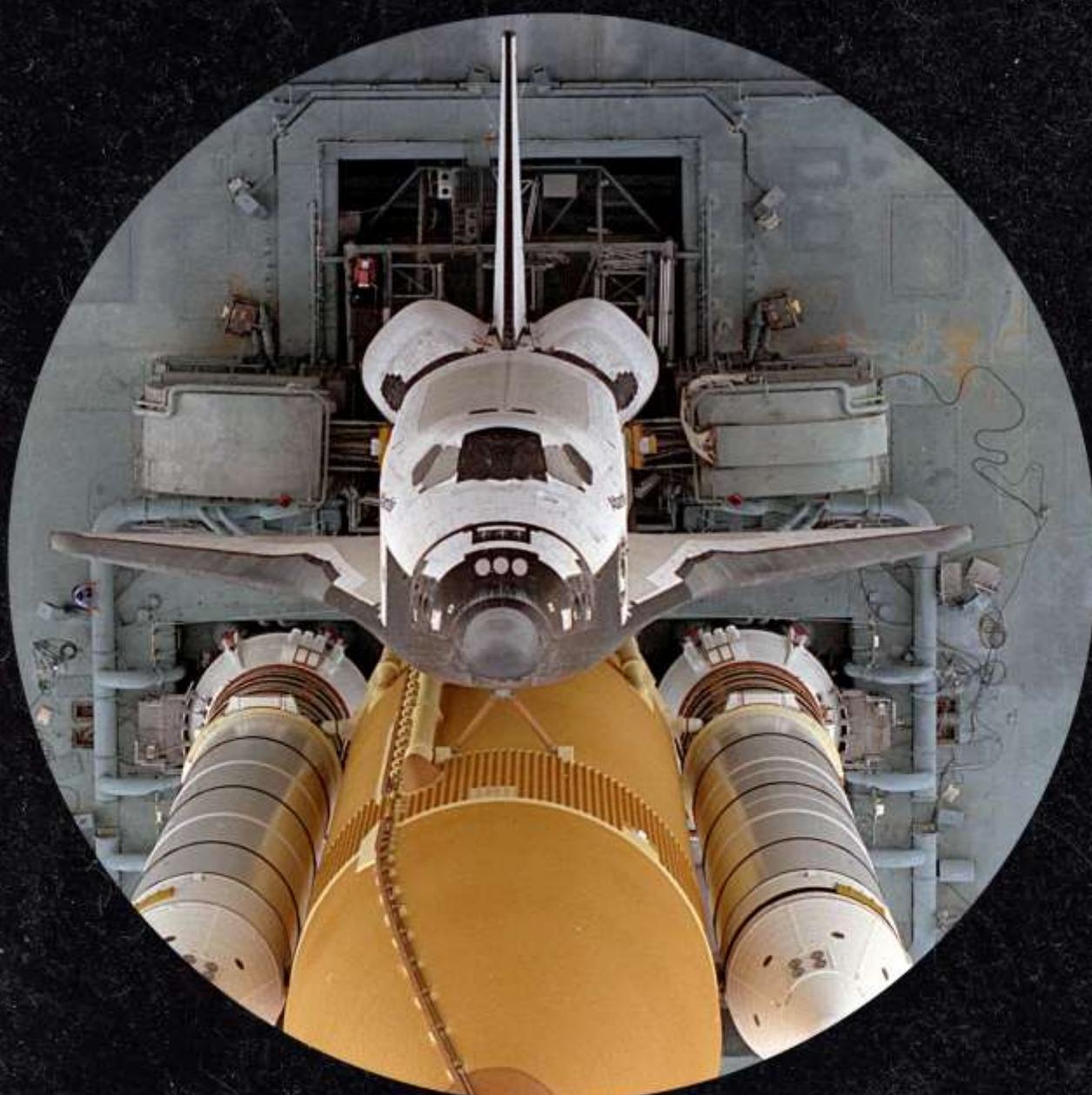
  return (
    <Watch milliseconds={today} />
  );
}

export default TodayTime;
```

זה יעבוד מעולה. ככה עובדים בפייתוח קומפוננטות מודרני. קומפוננטה שmobilia לkomponenta שmobilia לkomponanta. علينا לשאוף ככל האפשר לkomponantot קטנות בעלות תפוקדיות מוגבלת שנוכל למחזר שוב ושוב. למשל, בכל אתר גדול או אפליקציה יש ספריה פנימית של komponantot הצוגה שהמתכנים ממחררים שוב ושוב, וככה חוסכים זמן פיתוח. אבל כמו כל דבר בחיים, אם מגדמים במשהו זה עלול להוביל לביעות. כאשר אנו יוצרים יותר מדי komponantot, אנו עלולים ליצור בעיות ביצועים וזמן הרצה 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 פנימית כמו `setInterval`. אבל חלק גדול מהקומפוננטות אמורות לדעת לטפל בקלט מהמשתמש – כלומר פעולות ואירועים. אירועים הם חלק מהותי מג'או-הסקרייפט ומג'או-הסקריפט שרצ' בסביבת דפדפן, וריאקט יודעת לעבוד איתם יפה מאוד. על מנת לדעת איך עובדים עם אירועים, נזכיר איך אירוע DOM, כלומר אירועים טבאיים בג'או-הסקרייפט, עובדים.

AIROU DOM

בג'אווהסקרייפט, כאשר אנו עובדים עם אלמנטים טبויים של HTML, כמו למשל שוקון, אנו יכולים להציג להם אירועים. האירועים הללו נקראים אירועי DOM. להלן לוח של כמה אירועים נפוצים ולצדם דוגמאות:

דוגמה	שם אירוע
<button onclick="myFunction()">Click</button>	קליק
<select onchange="myFunction()">	שינוי
<input type="text" onblur="myFunction()">	יציאה מהאלמנט (יציאה מפוקו)
<input type="text" onfocus="myFunction()">	כניסה לאלמנט (פוקו)

האירועים הללו זמינים לאלמנטים הטבויים של HTML ואני מניח שאתם מכירים אותם. כדאי לשים לב שכל האירועים הללו כתובים באותיות קטנות ושאנו מעבירים אותם לאלמנט הטבעי במחזרת טקסט שרצה ברגע שהאירוע מופעל.

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

אירועים סינטטיים / ריאקטיביים

ב-XJS אנו יכולים להשתמש באירועים ריאקטיביים לאלמנטים שיעודו לעבוד איתם. הם דומים לאירועי DOM וגם מתנהגים כמעט כמויהם. מדובר בעצם באירועים עוטפים שיש להם אותו משקל 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-*camel case*, כלומר האות הראשונה של המילה השנייה תהיה אות גדולה באנגלית. למשל:

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

ובן שאין לא מעביר לאירוע מחרוזת טקסט אלא ביטוי בתוך סוגרים מסולסלים על מנת ש-XJS ידע להתמודד איתו. מעבר לזה, השימוש באירועים זהה לחלוון בריקט. כך, למשל, אם יש לי קומפוננטה שנקראת `MyInput` ובה אלמנט `input` של HTML (מדובר/alment שמאפשר לנו להקליד טקסט בתוכו), אני יכול להשתמש ב-`onChange` כדי להציג את מה שהמשתמש כותב בסיטייט ומשם לעשות אליו מה שבא ל'.

נדגים עם קומפוננטה פשוטה מאוד, שבה יש שוקון שכלי מה שנקליד בו יופיע על המסך שלנו. ניצור קומפוננטה שנקראת `InputViewer.js` ובה יהיו שני אלמנטים – `input` ו-`span` שיכיל את הטקסט. מה עם הטקסט? נשים אותו ב-`state` בדיק כפי שלמדנו.

```

import React, { useState } from 'react';

function InputViewer() {
  const [text, setText] = useState('');

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

export default InputViewer;

```

איך אנחנו מعتبرים את המידע מתחום ה-`text` בבדיקה בשליל זה אנחנו משתמשים ב-`onChange`, שלפי שמו אנו רואים שהוא אירוע ריאקטיבי. איך הוא עובד? באמצעות תוכנה שמצוידים לאלמנט `input`. התוכנה הזאת מקבלת פונקציה שמופעלת כאשר האירוע מופעל. למשל:

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

האירוע הוא `onClick` שמתתקיים כאשר אנו מקליקים על האלמנט. כאשר אנו מקליקים, הפונקציה שיש בתוך התוכנה `onClick` מופעלת. כזכור הפונקציה הזאת:

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

זו פונקציית חץ שפושט מופעלת עם `e`. מה זה? אירוע עצמוני, כלומר אובייקט עם מידע על האירוע. יש בಗ'אוהסקריפט כמה אירועים שאפשר להציגם לאלמנטים שונים.

הקוד הזה למשל:

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

הוא אירע onMouseEnter שמתקיים כאשר המשתמש עובר עמו הוכבר מעל האלמנט המדובר, במקרה זהה `onMouseEnter`, 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 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 React, { useState } from 'react';

function InputViewer() {
  const [text, setText] = useState('');

  function clickHandler(e) {
    console.log(e);
  }

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

export default InputViewer;
```

אולי זה יהיה קצת מפחיד מפונקציית חץ. נסו את הקוד הזה עכשו. הקliquו על ה-`input` ותראו איך פונקציית `clickHandler` מופעלת בכל לחיצה.

יש בג'אוהסקרייפט לא מעט אירועים - חלק מהם, כאמור, עובדים על אלמנטים מסוימים וחלק לא. האירועים הללו עובדים על אלמנט HTML בלבד, כמובן, לא על אלמנטים ריאקטיביים. אבל כרגע אין לנו בעיה, יש לנו אלמנט HTML פשוט. איך אנחנו משבירים את מה שאנחנו מקלידים בתוך ה-`input` אל הסטייט?

אנו נשתמש באירוע הריאקטיבי `onChange` שנקרא ושמופעל בכל פעם שיש שינוי בערך אלמנט ה-`input`. האירוע יפעיל פונקציה שנקראת `changeHandler` והוא תבצע>Action של הסטייט באמצעות `setText` שאותו הגדרנו באמצעות ההוק.

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

function InputViewer() {

  const [text, setText] = useState('');
  function changeHandler(e) {
    setText(e.target.value);
  }

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

export default InputViewer;
```

צרו את הקומפוננטה זו והללו בשדה ה-`input`. תוכלו לראות שהtekst מופיע מיד ב-`span`. מדוע אנחנו משבירים אל הסטייט רק את `e.target.value`? כי זה החלק באובייקט האירוע, כאמור, תוכנת האירוע מעבירה לנו, שמכיל את המידע שמעניין אותנו. באובייקט האירוע יש מידע רב ואתם מוזמנים לבדוק אותו עם הקונסולה או עם הדיבאגר, אבל ב-`onChange`, אבל ב-`onClick`, המידע הטקסטואלי שיש בשדה מועבר עם `e.target.value`.

אנו יכולים למשך כל התנהבות שאנו רוצים באמצעות אירוע. למשל, נניח שאנו רוצים שהמידע מתוך שדה ה-`input` יופיע בתוך ה-`checkbox` רק כאשר אנו מקליקים על כפתור. איך נעשה בזה דבר? ראשית, ניצור כפתור HTML פשוט שייהה לו אירוע `onClick`.

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

עכשו ניצור את הפונקציה שiodעת לעבוד עם האירוע. במקרה זהה מדובר באירוע קлик שלא מכיל מידע, אך איך אדע לבדוק מה יש בתוך ה-`input`? טוב, זה קל – יש לי את אירוע `changeHandler` שמכניס כל מה שאני מקליד שירות בשדה ה-`input` אליו משנתה סטייט שנקרה. אני צריך רק לנתק את ה-`text` מה-`span` ולחבר אותו לשנתה סטייט אחר (אקרא לו `viewText`), והלחיצה תגרום לסטטוס `viewText` מסתיט `text`.

```

import React, { useState } from 'react';

function InputViewer() {
  const [text, setText] = useState('');
  const [viewText, setViewText] = useState('');

  function changeHandler(e) {
    setText(e.target.value);
  }

  function clickHandler(e) {
    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 שבסוףו של דבר, האירועים בריאקט נראים למשתמש באופן כמעט זהה לאירועי ה-DOM הטבעיים ולא צריך להיבהל מהם. אנו מגדירים את האירועים הריאקטיביים לאלמנטי HTML שתומכים בהם כמעט כמו האירועים של ה-DOM, אך בכמה הבדלים קלים - שם הפונקציה שונה (onClick בעטיפה הריאקטיבית, onclick באירוע הטבעי) וכמובן מה שאנו מעבירים (פונקציית JSX שעתופה ב-{} בריאקט ומחזצת טקסט באירוע הטבעי). אם היינו משתמשים בג'אוויסקורייפט רגיל, הפעלת האירוע הייתה נראה כך:

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

אבל בغالל ה-JSX, אנו עושים זאת זה ללא הפעלה ובלי סוגרים מסולסלים.

יש עוד כמה שינויים קלים בין אירוע טבעי לאירוע ריאקטיבי. באירוע קליק ריאקטיבי, למשל, אנו בולמים את הפעופע לא באמצעות `return false` אלא באמצעות שימוש בפונקציה `stopPropagation`, אך לאណון לכך בפרק זה.

אבל איך אנו מזמנים אירועים לקומפוננטה ריאקטיבית? פה העניינים מסתובבים מעט, אבל ממש מעט. קומפוננטה ריאקטיבית יכולה להיות למשל קומפוננטה שמקילה `input`. משהו כזה:

```
import React from 'react';

function Input() {

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

export default Input;
```

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

אם אני רוצה להשתמש בקומפוננטה `Input` בדוגמה שהבאתי קודם, זה לכורה פשוט – אני רק אחליף את האלמנט `button` בקומפוננטה `Input`. משהו בסגנון כזה:

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

function InputViewer() {
  const [text, setText] = useState('');

  function changeHandler(e) {
    setText(e.target.value);
  }

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

export default InputViewer;
```

אבל אם לא תתעכלו ותנסו את הדוגמה בעצמכם, קלומר תעתייקו את `jsx` לתוכה הפROYיקט `Create React App`, תיצרו את קומפוננטת `InputViewer.jsx` ותציבו אותה באפליקציית הריאקט שלכם ב-`js/app`, תראו שזה לא עובד. למה?
כי קומפוננטות ריאקט לא מקבלות אירועים באופן טבעי.

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

```
import React from 'react';

function Input(props) {
  const changeHandler = props.onChange;

  return (
    <input onChange={changeHandler} type="text" />
  );
}

export default Input;
```

מה בעצם מתרחש כאן?
בקומפוננטה משתמשת ב-`props` אני לוקח את ה-`onChange` `props.onChange` ומעביר אותו לאלמנט ה-`input` הטבעי. כך, אם אני משתמש בקומפוננטה `Input` באופן הבא:

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

היא תעבד לי.
בואו נציג שוב עם הפתור באמצעות אחת הדוגמאות הקודמות. אני אוצר קומפוננטת כפטור ריאקטיבית שעוטפת כפטור HTML רגיל. שוב, זה נשמע מואלץ, אבל זו פרקטיקה נפוצה מאוד בספריות ובאפליקציות, ואם יצא לכם לעבוד או לראות קוד אמיתי בריאקט, תוכלם לראות את זה.

קומponent Ax.js תראה כר:

```
import React from 'react';

function Button(props) {

  const clickHandler = props.onClick;

  return (
    <button onClick={clickHandler}>Click</button>
  );
}

export default Button;
```

גם כאן, אני לוקח את כל מה שמעבירים לקומפוננטה ב-props ומעביר את זה הלאה, לאלמנט button הרגיל, של ה-HTML, שיעודו לעבוד עם אירועים. והשימוש? כרגע:

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

function InputViewer() {
  const [text, setText] = useState('');
  const [viewText, setViewText] = useState('');

  function changeHandler(e) {
    setText(e.target.value);
  }

  function clickHandler(e) {
    setViewText(text);
  }

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

export default InputViewer;
```

זה עלול להיות קצת מבלבל, אבל כדאי לזכור שמדובר בסופו של יומם ב-props שכבר מגדנו לעבוד איתם יפה מאוד ושהואם אנו מעבירים לקומפוננטה בקלות.

```

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, בקומponeנטה ריאקטית אנחנו חייבים להתייחס לכל אירוע שהוא רצים ב-props ולהעביר אותו אל האלמנט הטבעי.

תרגילים:

צרו קומפוננטה בשם `xs.js` שמכילה `div` בשם `MyDivContainer`. כאשר העבר עבר מעל הטקסט, תופיע המילה `active` באותו `div`.

רמז: יש צורך להציג שני מנהלי אירועים: הראשון הוא `onMouseOver`, שמנגיש לסתירת משתנה כלשהו ואת המילה `active`, והשני הוא `onMouseOut`, שמאפס את המשתנה.

פתרונות:

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

function MyDivContainer() {

  const [activeText, setActiveText] = useState('');

  function mouseoverHandler(e) {
    setActiveText('active');
  }

  function mouseoutHandler(e) {
    setActiveText('');
  }

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

export default MyDivContainer;
```

על אותו אלמנט יש לנו (שימו לב שאלמנט `div` הוא אלמנט HTML טבעי שיכל לקבל אירועים) שני אירועים שונים. האחד, `onMouseOver`, נכנס לפעולה כאשר העבר נמצא מעל האלמנט, והשני, `onMouseOut`, נכנס לפעולה כאשר העבר יצא ממנו. כל מה שנותר לי לעשות הוא ליצור את

הfonkcioot שפועלות בזמן שהARIOU מתרחש. הראשונה לוקחת סטייט שיצרתי ומכוינה לתוכו את המילה active והשנייה מאפסת אותו. אני מכניס את המשתנה שיש בסטייט ועובד אותו.

שימוש לב: אפקט מעבר עושים בדרך כלל עם CSS פשוט. כאן אנו עושים את זה עם ריאקט לשם התרגום. נוסף על כן, mouseOver הוא גם אירוע בעיתוי שיכל ליצור זליגות זיכרון.

תרגילים:

ארכיטקט המרכיב הביט בתרגיל הקודם וביקש שה-`div` יהיה קומponent ריאקט. קלומר שהקומponentה הקודמת תיראה כך:

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

function MyDivContainer() {

  const [activeText, setActiveText] = useState('');

  function mouseoverHandler(e) {
    setActiveText('active');
  }

  function mouseoutHandler(e) {
    setActiveText('');
  }

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

  export default MyDivContainer;
```

צרו את קומponentה `Div` ב-`axz.Div` כדי שהקוד שלעיל יעבוד.
رمز: `onMouseOver` ו-`onMouseOut` מועברים לקומponentה `Div` כ-`props` ועליכם להעביר אותם אל ה-`div` הטבעי של ה-HTML בקומponentה `Div`.

פתרונות:

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

function Div(props) {

  let onMouseOver = props.onMouseOver;
  let onMouseOut = props.onMouseOut;

  return (
    <div
      onMouseOver={onMouseOver}
      onMouseOut={onMouseOut}>
      My div
    </div>
  );
}

export default Div;
```

אפשר גם לkür את התהילך ולכתוב משהו כזה:

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

function Div(props) {

  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 `-props`, להעביר אותם לשניים ולהציג 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 הזה הֆונקציות המטפלות באירועים עוברות מאלמנט האב אל ה-props בקומפוננטה שלו, וברגע שהן ב-props אני יכול לשים אותן באלמנט שיש בקומפוננטה הפешוטה שלו.

תרגילים:

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

פתרונות:

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

function Counter() {

  const [count, setCount] = useState(0);

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

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

  function restartHandler() {
    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 להצמיד 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 כזה:

```
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 לשים Div או span כאבא. אבל זה כן בעייתי כי במקרה רבים אני לא רוצה שהיא לי אלמנט אב. למשל, אם אני יוצר קומפוננטות שמחלייפות tr, זהה אלמנט HTML שעוטף את השורה בטבלה. נניח שהוא כזה:

```
import React from 'react';

function TableRow() {

  return (
    <tr><td>Foo</td></tr>
    <tr><td>bar</td></tr>
  );
}

export default TableRow;
```

אני אהיה בבעיה. אני לא יכול לעוטף אותן ב-tr או td או span כי אם אני אציב אותן 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. המון פעמים 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 הינו יכולים להכניס אלמנט אב "ש��וף"! אז כן, יש אלמנט זהה שנקרא `React.Fragment`. האלמנט הזה יכול להיות במקום אלמנט אב והוא לא יודפס כלל. החל מהדוגמאות הבאות אני משתמש בו.

```
import React from 'react';

function TableRow() {
  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, זהה שלא מודפס ולא מתיחסים אליו. מקובל מאוד להשתמש בו בקומפוננטות בסיס ובקומפוננטות תצוגה.

פרק 13

USEFFECTS



useEffects

כפי שלמדנו בפרק על הסטייט, כSKUומפוננטה מסוימת משתנה, למשל בגל אירוע מסוים, היא נבנית מחדש על התצוגה שלה. הבנייה זו מוחדר נקראת רנדור. בהה נדגים באמצעות קומפוננטה שנבנתה באחד התרגילים בפרק על אירועים – הקומפוננטה Counter.jsx

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

function Counter() {

  const [count, setCount] = useState(0);

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

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

  function restartHandler() {
    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 שמשמעותו על התצוגה של הקומפוננטה זו (במקרה זה עם כפתורים, אבל זה יכול להיות שינוי שנוצר כתוצאה מ-`useState` שהשתנה), מתבצע רנדור מחדש. לא מעט פעמים אנו רוצים לדעת שהיota רנדור או רנדור מחדש של הקומפוננטה. בדיק בשביל זה אנו יכולים ליצור הוק מיוחד שיפעל פונקציה שאנו מעבירים לו.

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

הדרך הטובה ביותר להבין את זה היא באמצעות דוגמה:

```
import React, { useState, useEffect } from 'react';

function Counter() {

  const [count, setCount] = useState(0);

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

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

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

  function restartHandler() {
    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;
```

ראשתי הבהיר את useEffect כhook ב-import. הci קל ונחמד בעולם. השלב הבא הוא פשוט להשתמש בהוק זהה. אני קורא ל-`useEffect` ומעביר בו ארגומנט שהוא הפונקציה שתפעל.

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

מה שירוץ הוא:

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

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

בחרתי בפונקציית חץ פשוטה, אבל אין מניעה, כמובן, להשתמש בשם של פונקציה, שהוא בסגנון זהה:

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

useEffect(logEffect);
```

פונקציות חץ עלולות לבלבל, אבל צריך לזכור שהן בסופו של דבר פונקציות רגילות לכל דבר עם כמה תוספות חשובות, כמו למשל שמירה על `this`. זו הסיבה שcadai 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 `klass`. נרჩיב על כך בפרק על קומפוננטות מבוססות `klass`.

`useEffect` מכיל פרמטר נוסף שומולץ להשתמש בו. הפרמטר זהה ולוונטי כאשר אנו רוצים למדוד רנדור מחודש בעקבות שינויים `sdkrock` (ולא שינוי סטטייטים). זהו מערך של התכונות שאחריה יש לעקוב ורק אם הן משתנות, הפונקציה ב-`useEffect` תרוץ.

כך, למשל, אם יש לי קומפוננטה מסוימת ואני רוצה להקשייב לרנדור מחדש שלה שמתבצע רק בגלל שינוי props מסוימים, אני אוסיף ארגומנט שני את ה-props הרגולוניים:

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

export default function CountViewer(props) {
  const count = props.count;
  useEffect(() => 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 שהעבכנו כארוגומנט הראשון תרוץ בהתאם לפרמטרים המפורטים בארגומנט השני.

פרק 14

הומפוננטה ללא שם



קומפוננטה ללא שם

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

```
import React from 'react';

function Button(props) {

  const clickHandler = props.onClick;

  return (
    <button onClick={clickHandler}>Click</button>
  );
}

export default Button;
```

קראנו לפונקציה `Button` ואז ייצאנו אותה. זה סינטקס ולידי שnoch להבנה לאנשים שלומדים ריאקט לראשונה. אבל יש דרך נוספת לנוספת לכנות 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. `export` מקבל גם פונקציות חוץ אונומיות. כלומר כל קומפוננטה שכתבנו עד כה תעבור באופן מושלם גם אם נמיר אותה לפונקציית חוץ אונומית וניצא אותה. למשל, הפונקציה `Button` בהחלט תעבור כרגע אם היא תיראה כך:

```
import React from 'react';

export default (props) => {

  const clickHandler = props.onClick;

  return (
    <button onClick={clickHandler}>Click</button>
  );
}
```

זה נראה מפחיד, מבעית ואףילו מוזר לרוב המתכנתים שלא מרגלים בריאקט. אבל אם הגעתם לפרק הזה וקראתם היבט את כל הפרקים - וגם תרגלתם - 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`. הבעיה המרכזית היא שכל המפתחים של ריאקט לא יציג את שם הקומponentה ויקשה עליינו למצאו אותה.

פרק 15

עליזוב קומפוננטות



עיצוב קומפוננטות

יש כמה דרכים לעצב קומפוננטות בריект, חלון גם בעזרת ספריות עזר. בפרק זה אנו נתמקד בשתי דרכיהם 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 להספיק לעיצוב בסיסי.

בדפי אינטרנט בסיסיים יש לנו קובצי CSS שאנו מיבאים לדף ה-HTML, אבל באפליקציה ריאקט אנו יכולים להשתמש ב-import, וספרית וובפק תדאג לטוען את הקובץ ייחד עם הקומפוננטה.

ה-import כאן מבצע פשוטות 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 ה-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.jsx, שבנינו בפרק על אירועים.

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

function Counter() {

  const [count, setCount] = useState(0);

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

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

  function restartHandler() {
    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 וaz ניבא אותו. אנו נשים בראש הקומפוננטה את הטקסט:

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

את הקומפוננטה נציב, כרגע, ב-`App.js`, הקובץ הראשי של אפליקציית הריאקט שלנו. מחרג זה, כל שינוי שנעשה ב-CSS יוצג בקומפוננטה. על מנת לבדוק שהכל עובד, נציב בקובץ Counter.css את הטקסט הזה:

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

נסתכל בקומפוננטה ונראה שככל הרקע של הכפתורים הפך לאדום. חשוב מאד לציין: כיוון ש-`class` היא מילה שמורה בג'avascript, 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- הפרק הבא מלמד CSS בסיסי בלבד, אבל הוא יספק לכם את בסיס הידע הנחוץ כדי שתוכלו להמשיך הלאה.

CSS בסיסי

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

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

הселקטור `button` הוא הילקטור, התכונה `background-color` הוא הערך. אני מעניק את התכונות לכל האלמנטים שמתאים לסלקטור. במקרה הזה אני מעניק את תכונת צבע הרקע האדום לכל ה-buttons בלי יוצא מהכלל.

ילקטור

ילקטור 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 לכמה אלמנטים בקומפוננטה. אם אני רוצה לתת את התכונות האלו ל-myDiv, אני אתן אותן באמצעות הסימן נקודה - ".". כמובן, הסלקטור שלי יראה 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 myDiv וגם לכל ה-buttons, אני אכתוב אותם עם פסיק - "," מפריד ביניהם.

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

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

אפשר לשלב סלקטורים כדי להגיע לסלקטור מדויק יותר. למשל, אני מעניק את אותו הקלאש גם לחבר 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 קלאש myDiv באופן הבא:

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

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

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

```
<div id="myDivId">{count}</div>
az sелектор yihia:
```

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

משמעותו של id הוא ש-הו אמור להופיע פעם אחת בדף, באתר מבוסס קומפוננטות זה לא רעיון טוב בכלל. יש סלקטורים נוספים כמו סלקטורים לפי תוכנות או לפי תוכן – השמיים הם הגבול, אבל כאמור ניאלץ להסתפק בסלקטורים הבסיסיים. לא נדון בהיררכיה של סלקטורים, שהוא חלק ממשמעותי מאוד מ-.CSS

תכונות

הסלקטורים קובעים על מי נפעיל את התכונות. התכונות קובעות את העיצוב של האלמנטים: צבע, מידות ואףלו התחנוגות. התכונות נמצאות בתוך הסוגרים המסורלים שלאחור הסלקטור. הן מורכבות מהתcona ו מהערך שלה. למשל, שם התcona הוא צבע רקע – background-color – והערך הוא red. לכל תcona יש ערכים שאפשר לתת לה. למשל, התcona width תקבל ערכים בפיקולים או ביחידות אחרות.

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

אפשר לתת כמה תכונות שרצים, ללא הגבלה, כל עוד מקפידים שהיה סימן נקודה-פסיק (;) לאחר כל תcona.

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

הינה רשימה קצרה של תכונות והערכים שלהן. בעולם האמיתי יש כמובן הרבה יותר תכונות, אבל כאן נדרש להסתפק ברשימה קצרה:

הערכים שלהם	שם התכונה	תיאור התכונה
מספר + אק, למשל: 100px	height	גובה
מספר + אק, למשל: 100px	width	רוחב
שם הצבע, למשל red, או ערך הצבע בהקס' למשל: #fffffff	color	צבע טקסט
שם הצבע, למשל red, או ערך הצבע בהקס' למשל: #fffffff	background-color	צבע רקע
מספר + אק ארבע פעמים. הערך הראשון קובע את השולטים העליונים, השני את השולטים מימיין, השלישי את התחתונים והאחרון את השולטים משמאלי. למשל: .10px 10px 10px 10px	margin	שורדים
right או left	text-align	כיוון טקסט בתחום האלמנט

עכשו, כשייש לכם את היסודות לבניית CSS בסיסי, תוכלו ליצור את קובץ ה-CSS שלכם ולבצע להם `import` או `לחילופין` לעצב את הקומפוננטה בדרכים אחרות. יש כמה וכמה דרכים נוספות לעיצוב קומפוננטה בריאקט, אבל כל הדרכים הללו נשענות על ספריות צד שלישי. לריאקט עצמה אין דעה בנוגע לדרכי העיצוב, ובדוקומנטציה הרשמית שלה אפשר למצוא את הדרך שפורטה פה. יש דרך נוספת לעיצוב קומפוננטות בשם CSS `inline`, אך היא אינה מומלצת על ידי הדוקומנטציה ופוגעת **בביצועים**.

פרק 16

הלאס ריאקטוי



קלאוס ריאקטיביים

עד כה, כל הקומפוננטות שעבדנו עליון היו פונקציות. הקומפוננטות האלו נקראות באנגלית **functional component** והן משתמשות בהוקים על מנת לנחל את הסטיט ו גם לדעת מתי הקומפוננטה רונדרה. אבל יש גם קומפוננטות אחרות, קומפוננטות מבוססות **קלאוס**.

אם הספר זהה היה נכתב בתקופה של ריאקט ה'יתה בגרסה 15, סביר להניח שהקומפוננטות האלו היו תופסות בו נפח גדול יותר, אבל החל מגרסה 16.8 השימוש בקומפוננטות מבוססות פונקציות הפך לפחות יוטר. כרגע אפשר לראות מעבר של יותר ויותר מפתחים ושל ריאקט עצמה לכיוון הקומפוננטות מבוססות הפונקציה. אבל חשוב לדעת שיש קומפוננטות מבוססות קלאוס בריאקט ولو רק בגלל היכולת שלהם לעבוד עם קוד ישן יותר (לאגטי קוד, Legacy Code). נוסף על כך, עדין יש שליטה טובה יותר בمعالג החיצים הריאקטיביים בקלאסים, אבל הדגש הוא על עדין.

از איך זה נראה? פשוט מאד – עם קלאוס. קלאוס (Class) הוא חלק מג'אוועהסקרייפט עצמה ומדובר בסינטקס שונה מפונקציה. לקלאוס יש כמה חלקים – ראשית, יש לו **קונסטרוקטור** (constructor), הפונקציה הבנאית, שפועלת מיד בהתחלה ושאליה אני גם מקבל את **ה-props**. הקוד בקונסטרוקטור מתבצע ברנדור הראשוני בלבד. שנית, יש לו **render**, שיש בו מה שהקומפוננטה מחזירה.

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

```
import React from 'react';

function Welcome() {

  return (
    <div>Hello World!</div>
  );
}

export default Welcome;
```

המקבילה שלה ב-`ES6` תיראה כך:

```
import React from 'react';

class Welcome extends React.Component {

  render() {
    return <div>Hello World!</div>;
  }
}

export default Welcome;
```

אפשר לראות כאן שמתודת `render`, שיש לנו אותה כיוון שהקלואו שלנו יירש מה-`React.Component`, אחראית על החזרת ה-`JSX`. עד כאן אין הבדל משמעותי בין קומפוננטת קלואו לקומפוננטת פונקציה, שאותה אנו מכירים.

הענינים נוספים פשוטים גם אם אנו מתחילה להשתמש ב-`props`. ה-`props` ב-`ES6` נמצאים על גבי הסkop של הקומפוננטה, כלומר ב-`this.props`. בכל מקום נתון בקומפוננטת הקלואו אני יכול להשתמש ב-`this.props` ולקבל גישה ל-`props`. כדי להדגים, הבה נניח שאנו רוצים להוסיף את שם המשמש ולקבל אותו מה-`props`. בקומפוננטת פונקציה נעשה משהו כזה:

```
import React from 'react';
```

```
function Welcome(props) {
  return (
    <div>Hello {props.name}!</div>
  );
}

export default Welcome;
```

בכלאוס אנו ניגשים אל ה-`props` באמצעות `this`. אבל למעשה העובדה הזאת, אין שינוי מהותי. על מנת למש את אותה התנהוגות בדיק עם קלאוס, אני אכתוב את הקוד הבא:

```
import React from 'react';

class Welcome extends React.Component {
  render() {
    return <div>Hello {this.props.name}!</div>;
  }
}

export default Welcome;
```

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

הבה נניח שאנו רוצים ליצור סטיט עם ברכה – כולם לקלבל את ה-`props.name`, ליצור ברכה ולהכניס אותה לסטיט. אם אני ארצה לעשות את זה עם פונקציה, רק לשם הדוגמה, אני אעשה משהו כזה:

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

function Welcome(props) {

  const [greeting, setGreeting] = useState(`Hello ${props.name}`);

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

export default Welcome;
```

איך נשימוש בסטייט בקלאס? במקרה זהה נצטרך להשתמש בקונסטרוקטור, שבו נאתחל את הסטייט ונכניס לתוכו ערך ראשון. הסטייט הוא אובייקט שיושב בסקוופ של הקומפוננטה (כלומר תחת ה-this). אין לנו הוק במקרה זהה וכשאנו רוצים להשתמש בסטייט בקלאס, אנו פשוט מגדירים אותו בקונסטרוקטור. קומפוננטת הקלאס שלנו תיראה כך:

```
import React from 'react';

class Welcome extends React.Component {
  constructor(props) {
    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 קרייטית כיון שהיא קוראת לקונסטרוקטור של React.Component. זו לא המצאה של ריאקט אלא חלק מהסינטקס של ג'אווהסקריפט. בלי השורה הזו, הקונסטרוקטור שלכם לא יעבד כשרה.

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

אין לנו הוקים וaned לא מגדירים פונקציה שניי סטיט. הפונקציה שאנו משתמשים בה לשינוי הסטיט היא `setState` שמקבלת אובייקט סטיט. השימוש שלה יראה כך:

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

זה נראה מורכב יותר מאשר המימוש הנוכחי והנק' יותר של הפונקציה, וזה נכון. מכיוון שיש לנו קלאו, אנחנו נכנסים גם לבעיות של קונטקט.

הבה נדגים על ידי המרת הקומפוננטה זו, שהיא פונקציה משתמשת באירועים, לפחות:

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

function Counter() {

  const [count, setCount] = useState(0);

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

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

  function restartHandler() {
    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 נזכור ליצור קונסטרוקטור ולשנות את הסטייטים, אבל יש לנו גם אירועים, ו/events אנו חייבים להעביר קונטיקסט של `this`. אחרת, בשעת הפעלת האירוע תעבד הפונקציה בסקוופ אחר (הסקוופ של החלון) ולא תהיה לה גישה לסטיט.

```
import React from 'react';

class Counter extends React.Component {

  constructor(props) {
    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 לשים לב היטב לשורות האלו בكونסטרקטור:

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

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

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

הסקופ הוא לא של הקלאס. על מנת Lagerom לאירוע לעבוד בתוך הקלאס עם נג Ishot למשתנים, למתודות וכמובן לסטיט, יש צורך להשתמש ב-bind. ה-bind גורם 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 עכשוו על קלאוס ועל פונקציה:

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

קל לראות שקלאס הוא מסובך יותר מבחןת כמות 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 קלאוס. לפיכך, הדוגמאות ימשיכו להיות בפונקציות.

מעגל החיים ב-**KLАО**

אחד היתרונות הגדולים של שימוש בקומפוננטה מבוססת KLАО על פניו פונקציה הוא יכולת השיליטה הגדולה יותר במעגל החיים שלה. בעוד בפונקציה יש לנו רק את הhook Effect, שפועל כאשר הקומפוננטה מתורננדרת בפעם הראשונה וכאשר היא מתורננדרת מחדש, ב-**KLАО** יש לנו שליטה מלאה יותר במעגל החיים וק"י מות לא פחות מאשר מתודות לעשות זאת.

componentDidMount

מתודה המופעלת מיד לאחר שהקומפוננטה נטעןת לתוך-DOM. זה מקום מעולה לבצע קרייה ל-**API** ופעולות נוספת הקשורות לאתחול הקומפוננטה (כמו לוגינג) או להירשם לAIROOUIPs, כפי שנראה בדוגמה שבמהלך הפרק.

componentDidUpdate (prevProps, prevState, snapshot)

מתודה המופעלת ברגע שה-**props** מתעדכנים או ה-**setState** מופעל. אנו מקבלים גישה ל-**prevProps** והקודמים ול-**prevState** הבאים (אחרי העדכון). הפרמטר השלישי הוא נדרי יותר ורלוונטי לשימוש ב-**getSnapshotBeforeUpdate** ולא נדון בו.

שימוש לב: בניגוד ל-**componentDidMount**, שנקרא רק פעם אחת במעגל החיים של קומפוננטה, **componentDidUpdate** נקרא לאחר כל פעם שמתבצע רנדור (חוץ מהפעם הראשונה, שבה **componentDidMount** מטפל). כדי גם לשימוש לב שמןיני שהמתודה הזאת מופעלת כתוצאה מ-**setState**, אם נשים בתוכה עוד **useState** בלי תנאי, עלולה להיות לנו לולה אינסופית.

componentWillUnmount

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

shouldComponentUpdate (nextProps, nextState)

מתודה שבה אנו יכולים לקבוע אם אנו בכלל רוצים שהקומפוננטה תתרנnder מחדש. היא מופעלת כאשר ה-**props** או הסטייט מתעדכנים. המתודה הזאת אמורה להחזיר **true** אם אנו רוצים רנדור מחדש או **false** אם אנו לא רוצים רנדור כזה. היא נדירה יחסית ומאפשרת לנו גישה אל ה-**props** שהತעדכנו, ואל הסטייט החדש. היא מתרחשת לפני **componentDidUpdate** כאמור, אם היא מחזירה **false**, איז **componentDidUpdate** לא תרוץ כיון שהקומפוננטה לא תרונדר. התכוון העתידי של ריאקט הוא לאו דווקא להפוך את זה להוראה אחידה אלא להמליצה למנוע את הרנדור של ריאקט.

היא נדירה יחסית כי ברוב המוחלט של המקרים אנו נשורך על המנוע של ריאקט שיקבע אם קומפוננטה תרונדר או לא. אנו משתמשים בה כאשר נרצה לשפר ביצועים כיון שאם חוסכים בrndor, מקבלים ביצועים משופרים לכל הקומפוננטות.

getDerivedStateFromError (error)

מתודת שמו פעולה כאשר אחת הקומפוננטות הבנות זורקת שגיאה ויכולת לעדכן את הסטיטו בעקבות השגיאה זו (ולא לבצע פעולות נוספות לכך מאשר לשנות הסטיטו). מה שהיא מחייבת מעדכן את הסטיטו.

componentDidCatch (error, info)

בדומה ל-`getDerivedStateFromError`, גם המתודה זו מופעלת כאשר אחת ממתת-הקומפוננטות זורקת שגיאה, אך היא יכולה גם לקרוא לפונקציות אחרות (כמו `log`). חלון מיועדות לשימוש קצה, אבל חלון שימושיות מאוד בקומפוננטות מסוימות. כך, למשל, בקומפוננטות שיש בהן `interval`, מומלץ מאוד להשתמש דווקא בקומפוננטות כאלה כי אפשר לנתקות את ה-`interval` אחר כך ולהקל על הזיכרון. אם לא נעשה זאת זה, ה-`interval` ימשיך לרווח גם אם הקומפוננטה נמחקה או אם עברנו למקום אחר, וזה יכול לגרום זליגת זיכרון ובעיות ביצועים.

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

```
import React from 'react';
class TodayTime extends React.Component {

  constructor(props) {
    super(props);
    this.interval = setInterval(() => {this.upTime()}, 1000);
    this.state = { time: Date.now() };
  }

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

  componentWillUnmount() {
    clearInterval(this.interval); // Cleaning phase
  }

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

export default TodayTime;
```

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

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

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

עיהון 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.

תרגילים:

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

פתרונות:

```
import React from 'react';

class ClickCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      clicksNumber: 0,
    };
    this.clickHandler = this.clickHandler.bind(this);
  }

  clickHandler() {
    this.setState({ clicksNumber: this.state.clicksNumber + 1
  });
  }

  render() {
    return <div>
      {this.state.clicksNumber}
      <button onClick={this.clickHandler}>+</button>
    </div>
  }
}

export default ClickCounter;
```

אין כאן מושהו שאינכם מכירים. את הסטייט אנו מגדירים כבר בקונסטרקטור ואנו מתחלים אותו כ-0, ייצרנו handler לקליק שמעלה את הסטייט ב-1. כיוון שהAIROU פועל בסקוופ אחר, אנו חייבים לקשור אליו את הסקוופ שלנו באמצעות:

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

אחרת נקבל שגיאת TypeError: Cannot read property 'setState' of undefined בכל פעם שנפעיל את AIROU, כיון שהוא-this שיש באIROU אינו ה-this של הקלאס.

תרגילים:

מנהל המוצר של בזק פנה אליכם בבקשתה ליצור קומפוננטה שמציגה התקפות סייבר. המספר שלן מתחילה ב-0,000 ועולה ב-10 בכל שנייה. עליכם למש זאת עם קלואס כיוון שאתם צריכים להשתמש ב-setInterval ויש צורך לנוקות אותו עם clearInterval.

פתרונות:

```
import React from 'react';

class CurrentCyberAttackCounter extends React.Component {
  constructor(props) {
    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 ב-`setInterval` אם יש צורך להשתמש בניקיון אחריו שהקומפוננטה נעלמת. ראשית, אנו משתמשים ב-`setInterval` כרגע. כיוון שיש לי צורך בזיכרון פנימי, אני יודע שאני צריך סטייט ואני מגדיר סטייט כבר בקונסטרוקטור באמצעות `this.state`.

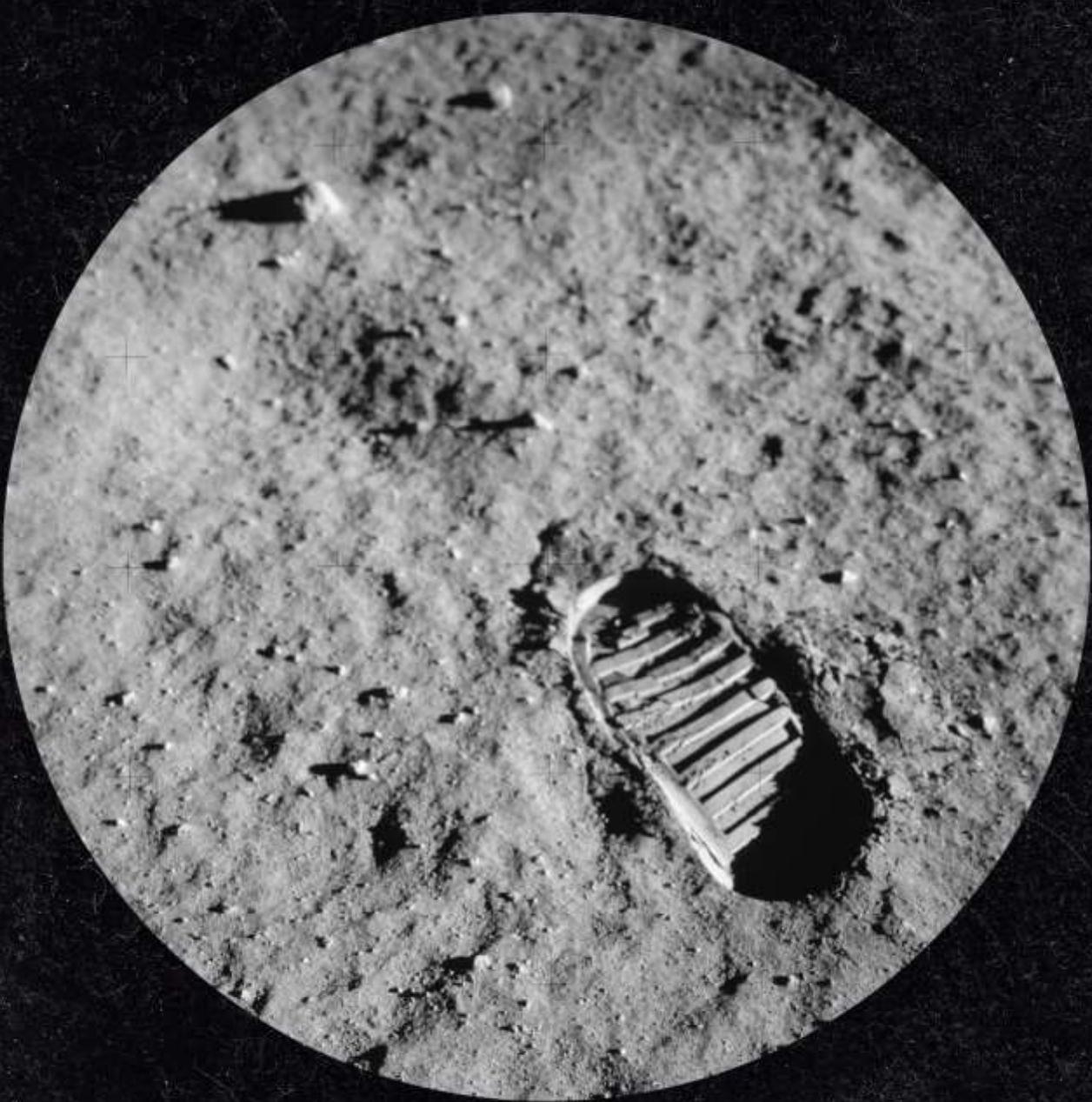
הפעולה השנייה היא להגדיר `interval` שיפעל בכל שנייה, ואת זה אני עושה במתודת מעגל החיים `componentDidMount`, שעובדת מיד לאחר שהקומפוננטה מופעלת. אני מייצר `interval` וויצר לו רפרנס במשתנה שנמצא בסkop של הקומפוננטה. מה שה-`interval` עושה הוא לעדכן את הסטייט ב-`+10`. על מנת שתהיה לו גישה לסטטיט, אני מפעיל:

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

את הnickion אני עושה בקומפוננטה מעגל החיים `componentWillUnmount`, ואת הפלט אני מגדר `render`.

פרק 17

שימוש בהומפוניות סמלודות אחרים



שימוש בקומפוננטות ממקורות אחרים

אחת מהחוויות הגדולות של ריאקט היא שימוש בספריות אחרות של קומפוננטות. כך בעצם אנחנו לא נדרשים לבנות מאפס את האפליקציות שלנו אלא פשוט להשתמש בהשתמש בקומפוננטות בקוד פתוח שאחרים בנו וכך אנחנו יכולים להגיע לתוכאות טובות מאוד בזמן קצר יותר. יש בחוץ אינספור קומפוננטות ועודפי קומפוננטות שקל ללקח וליצור איתם אפליקציות מורכבות ביותר ויפות מאוד מבהינה ויזואלית.

הספריות האלה באוט כabilות תוכנה שאנו יכולים להתקין בפרויקט שלנו, שיצרנו ב-App. הפרויקט הזה מכיל לא מעט כabilות תוכנה, כמו babel למשל, והcabilitות האלה מנוהלות על ידי Node.js (ג'אויסקripsט בסביבת השרת) שנקרה וקוח. לא נדרש לדעת Node.js כדי להשתמש בו-וקח ולהתקין כabilות תוכנה (או להטיס אותו).

על מנת להתקין כabilות תוכנה הכוללת קומפוננטות, אנו נדרשים לשורת הפוקודה שעלייה למדנו בפרק Terminal על בניית סביבת עבודה מתקדמת. אנו נפתח את Visual Studio Code או נפתח את לשונית New Terminal על Linux. יפתח לנו ממשך של שורת פוקודה בתחתית העורך ונוכל לבצע התקינה באמצעות cmd.

חלופין, אם אין לכם, מסיבה מסוימת, Visual Studio Code, אתם יכולים, אם יש לכם מחשב מבוסס חלונות, להגיע למצב cmd ואז לנoot באמצעות הפוקודה cd אל תיקיית הפרויקט שלכם. גם שם אפשר לבצע התקינה צזו.

כדי להתקין קומפוננטות ממקומות אחרים אנחנו צריכים לדעת אילו קומפוננטות אנו רוצים. יש אלפי אוספי קומפוננטות וקומפוננטות שונות. מקור טוב לחפש הוא גול. כאן נתרgal באמצעות React UI Material – אוסף קומפוננטות בעיצוב חברת גול שיצר המון קומפוננטות בסיסיות בריאקט. העיצוב הזה נפוץ מאד ולא כמעט אפליקציות ווביית אתרים משתמשים באוסף הזה. כתובות הפרויקט היא:

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

מיד בדף הראשון נבחר ב-get started ונראה שיש לנו הוראות מדויקות להתקנה עם מקום. נעקוב אחריהן. בחלון הטרמינל שלנו נקליד:

```
npm install @material-ui/core
```

הפקודה הזאת משתמשת ב-PLACEHOLDER על מנת להויריד ולהוסיף לפרויקט באופן אוטומטי את ספריית React UI על קומponeנטות שלה.

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

```
PS C:\Users\barzik\local\my-app> npm install @material-ui/core
npm WARN @typescript-eslint/well-known-plugin@1.1.0 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 but none is installed. You must install peer dependencies yourself.
npm WARN tslint@8.17.1 requires a peer of typescript@>2.8.8 || <= 3.2.0-dev || >= 3.3.0-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\node-sass\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@2.0.7 (node_modules\fs-events):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.0.7: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ @material-ui/core@1.6.1
added 31 packages from 41 contributors and audited 963760 packages in 17.413s
```

עכשיו אפשר להשתמש בקומponeנטות האלה.

חשוב: בשנים האחרונות הפכו APIs Node.js ו-PLACEHOLDER לשימושם בכל הקשור לסבירות פיתוח, וכל הספריות המודרניות משתמשות ב-PLACEHOLDER להתקנה ולניהול. ספר זה אינו מלמד מקום אך הוא קל להפעלה. להלן כמה פקודות שימושיות בטרמינל. יש להקליד אותן מהתיקייה שבה הפרויקט נמצא:

הפקודה	תיאור הפעולה
npm install PACKAGE_NAME	התקנה של חבילת תוכנה
npm uninstall PACKAGE_NAME	הסרת התקנה של חבילת תוכנה
npm ls PACKAGE_NAME או בדיקה של הקובץ package.json	בדיקות אם חבילת תוכנה מותקנת אצלם

על מנת להשתמש בקומponeנטות אלו צריך רק להשתמש ב-import, כמו בכל קומponeנטה. ההבדל היחיד הוא שהוא לא נדרש לבצע import מרכיב עם יבוא מתיקייה אלא import עם שם החבילה בלבד.

אחת הקומponeנטות היא קומponeנטה כפטור בשם Button, שבעצם יוצרת כפטורים מעוצבים. הנה נשתמש בה עם הקומponeנטה Counter שבנו בעבר:

```

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

function Counter() {

  const [count, setCount] = useState(0);

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

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

  function restartHandler() {
    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 תריצו את הקומפוננטה הזו, תוכלו לראות שהכפתורים הם כפתורים רגילים ומכוונים. הנה ניצור כפתורים מעוצבים יותר. איך? יש בספריה של UI React Material כפתורים יפים הרבה יותר. שם הרכיב של הכפתור הוא Button. אנו מיבאים אותו באמצעות:

```
import Button from '@material-ui/core/Button';
```

ומשתמשים בו בדיקן כמו 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` משפיעים על מראה הcptורום. בטעות של הקומפוננטה של cptורם, וכל קומפוננטה אחרת, יש הסברים על ה-`props` שאפשר להעביר ועל איך הם מושנים את המראה של הקומפוננטה.

אפשר, כמובן, להשתמש בכמה קומפוננטות שרצוים באותה קומפוננטה, בדיק כמו בקומפוננטה רגילה!

```
import React, { useState } from 'react';
import TextField from '@material-ui/core/TextField';
import Button from '@material-ui/core/Button';

function InputViewer() {
  const [text, setText] = useState('');
  const [viewText, setViewText] = useState('');

  function changeHandler(e) {
    setText(e.target.value);
  };

  function clickHandler(e) {
    setViewText(text);
  };

  return (
    <div>
      <span>{viewText}</span>
      <TextField onChange={changeHandler} />
      <Button onClick={clickHandler}>Click me</Button>
    </div>
  );
}

export default InputViewer;
```

יבוא כמה קומפוננטות

אם אני מיבא כמה קומפוננטות מסוימת ספרייה, מקובל להשתמש בסינטקס מעט שונה ליבוא. במקום:

```
import TextField from '@material-ui/core/TextField';
import Button from '@material-ui/core/Button';
```

כותבים:

```
import { TextField, Button } from '@material-ui/core';
```

אנו פשוט משתמשים בהמרה של ג'אוΗסΚריפט - זו צורת כתיב שונה לאותו הדבר.

חשוב לציין שההכוח האמייתי של ריאקט – השימוש בספריות קומפוננטות בלבד, שנוננות לנו כוח אדיר רק באמצעות התקינה פשוטה של ספריות קוד פתוח ושימוש קל. באתר של UI React 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 התקינה מדיוקנות:

<https://devexpress.github.io/devextreme-reactive/react/grid/>

אבל לנו נדגים גם פה – ראשית יש להתקין את הקומפוננטה באמצעות kompack או הפיק ban:

```
npm install @devexpress/dx-react-core
npm install @devexpress/dx-react-grid
npm install @devexpress/dx-react-grid-material-ui
npm install @material-ui/icons
```

השלב הבא הוא פשוט... להשתמש בה!

```

import React from 'react';
import { Grid, Table, TableHeaderRow } from '@devexpress/dx-react-
grid-material-ui';

function TableViewer() {

  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 לעזור לכם העבודה ומאפשרים לכם לקבל תוצאות מהירות ויפות כמעט מיד. שווה להתחיל להתעניין ולנסות, למשל, את UI React Material. יש הרבה דמואים ודוגמאות קוד מעניינות שדרכם אתם יכולים ליצור תוך מהם כמעט מושך.

מודולים חסרים

לעתים, כשתנסו קומפוננטות מסוימות אחרות, אתם עלולים להיתקל בשגיאה נוכח:

```
Module not found: Can't resolve '@material-ui/icons/ChevronLeft' in  
C:\PROJECT
```

או בשגיאה דומה. השגיאות הללו מתקבלות כאשר המודול פשוט לא הותקן על ידי ווקט. במקרה זהה פשוט כדאי לגלל את התוצאה או להתקן את הספרייה. בשגיאה זו הספרייה החסורה היא `@material-ui/icons` והבעיה נפתרת על ידי הקלה בטרמינל של השורה:

```
npm install @material-ui/icons
```

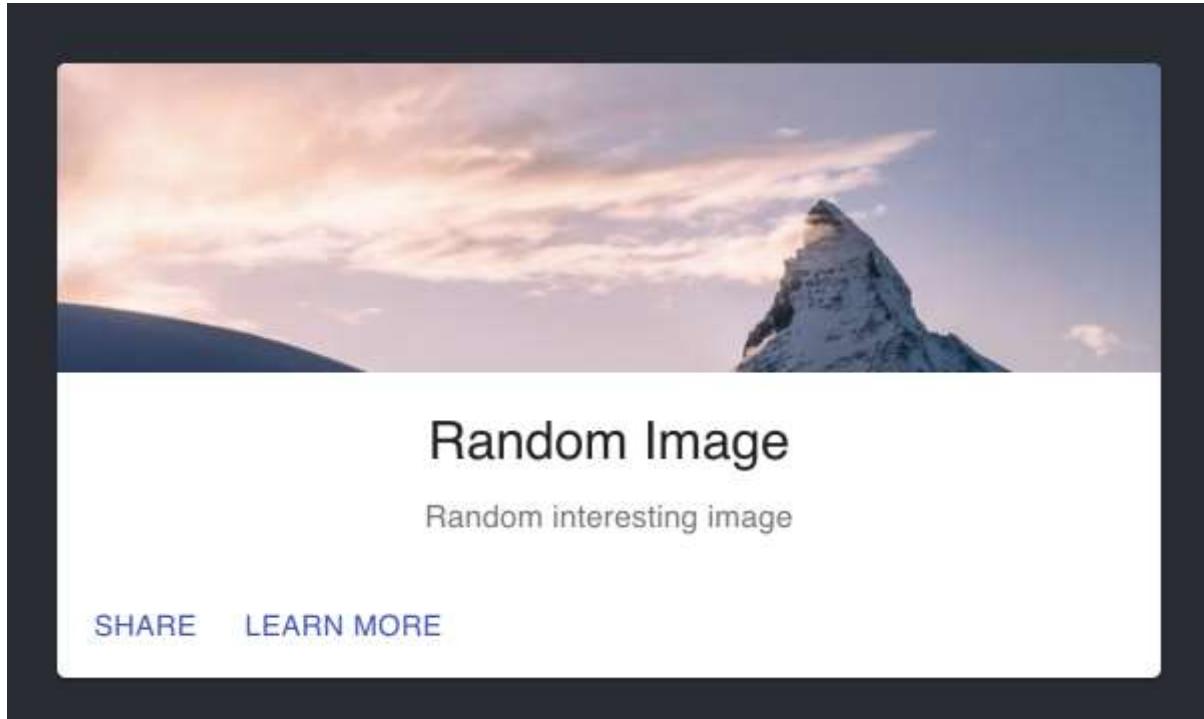
היא מתקינה את הספרייה החסורה שצרי לשימוש בה.

תרגילים:

באמצעות הרכיב `card` שיש ב-UI React Material media card וכן באמצעות הקישור לתמונה רנדומלית:

<https://picsum.photos/id/866/700/400>

צרו את הרכיב זה:



רמז: התיעוד והדוגמאות לכרטיס נמצאים בכתובת זו:

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

פתרונות:

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Card from '@material-ui/core/Card';
import CardActionArea from '@material-ui/core/CardActionArea';
import CardActions from '@material-ui/core/CardActions';
importCardContent from '@material-ui/core/CardContent';
import CardMedia from '@material-ui/core/CardMedia';
import Button from '@material-ui/core/Button';
import Typography from '@material-ui/core/Typography';

const useStyles = makeStyles({
  card: {
    width: 500,
  },
  media: {
    height: 140,
  },
});

export default () => {
  const classes = useStyles();

  return (
    <Card className={classes.card}>
      <CardActionArea>
        <CardMedia
          className={classes.media}
          image="https://picsum.photos/id/866/700/400"
          title="Random Image"
        />
      <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>
);
}

```

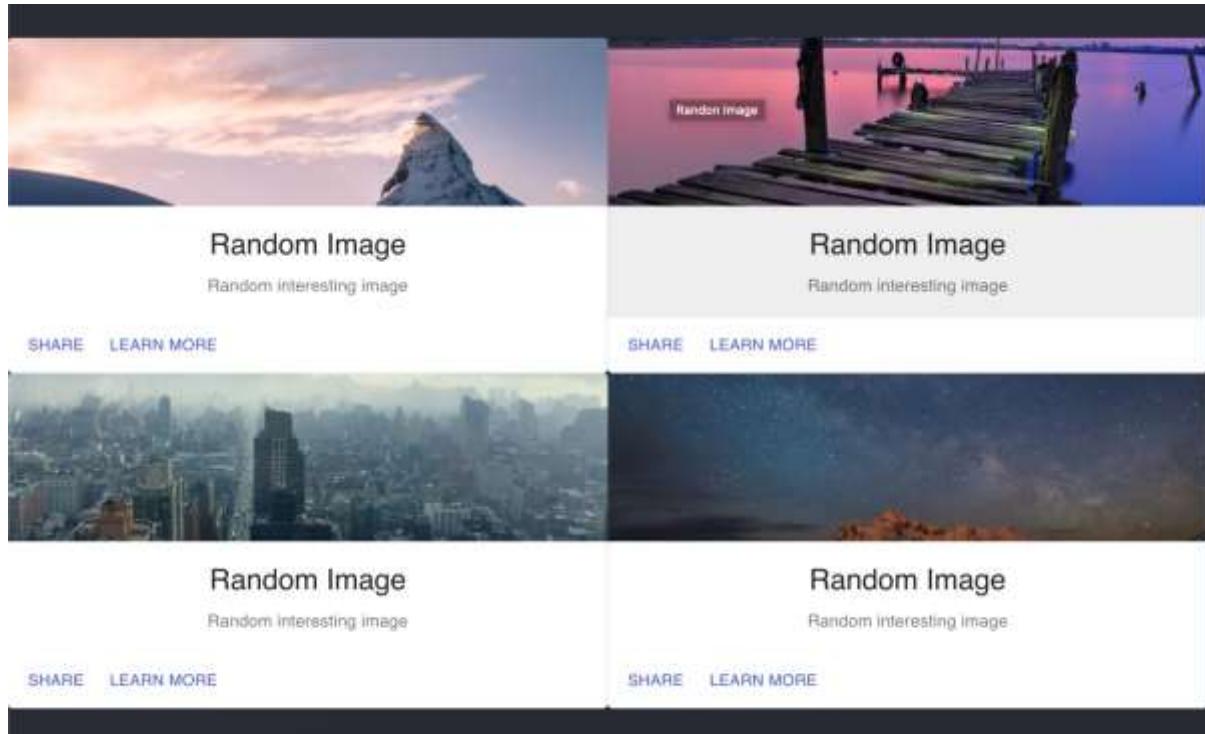
אין כאן גאונות גדולה, פשוט העתקתי את הדוגמה המופיעה בדוקומנטציה כדי ליצור את מה שרציתי ליצור. במקרים הללו אין מה לשבור את הראש ולהתאמץ, אלא צריך פשוט לקחת את מה שיש בדוקומנטציה.

כדי לשים לב שהדרך של UI React Material לקביעת עיצוב היא דרך שונה המשמשת ב-`useStyles`, זהה HOC. לא צריך להבין איך זה עובד על מנת להשתמש בזה, פשוט לזרום עם מה שיש בדוקומנטציה. על HOC נלמד בהרחבה בפרק הבא.

את הקוד הזה אפשר לשימוש ב-`RandomImageCard.jsx`, וזה להשתמש בקומפוננטה זו איפה שרוצים או לדאוג להעביר לה `props` שיקבעו את התמונה השונה.

תרגיל:

באמצעות הרכיב הקודם, צרו ארבעה כרטיסים כאלה:



עם התמונות שב קישורים 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 בדיק כפי שלמדנו.

פתרונות:

אנו צריכים ללקח את הקומפוננטה הקודמת, זו שהעתקנו מהדוקומנטציה, ופשוט לשנות אותה כך שתדע לקבל `props`. השינוי הזה הוא פשוט: הקומפוננטה מעבירה `props` והם נכנסים במקום כתובת התמונה בקוד הזה:

`<CardMedia`

```
  className={classes.media}
  image="https://picsum.photos/id/866/700/400"
  title="Randon Image"
/>>
```

אמנם הקומפוננטה נראהיה מורכבת, אבל השינוי הזה אינו מורכב:

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Card from '@material-ui/core/Card';
import CardActionArea from '@material-ui/core/CardActionArea';
import CardActions from '@material-ui/core/CardActions';
importCardContent from '@material-ui/core/CardContent';
import CardMedia from '@material-ui/core/CardMedia';
import Button from '@material-ui/core/Button';
import Typography from '@material-ui/core/Typography';

const useStyles = makeStyles({
  card: {
    width: 500,
  },
  media: {
    height: 140,
  },
});

export default (props) => {
  const classes = useStyles();
  return (
    <Card
      className={classes.card}
      image="https://picsum.photos/id/866/700/400"
      title="Randon Image"
    />
  );
}
```

```
<Card className={classes.card}>
  <CardActionArea>
    <CardMedia
      className={classes.media}
      image={props.imageSrc}
      title="Random Image"
    />
    <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 האלו ולעשות את זה ארבע פעמים בקומפוננטה מסוימת, למשל ב-**MyContainer**:

```
import RandomImageCard from './RandomImageCard';
import { Grid } from '@material-ui/core';

function MyContainer() {

  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. תוכל לבדוק אותה בדוקומנטציה. אם השתמשתם ב-CSS זה מעולה, אבל קל יותר להשתמש בקומפוננטה מסודרת שמישה כבר יצר.

כל שנוטר לי הוא לקחת את קומפוננטת MyContainer ולהציב אותה איפה שאני רוצה, למשל ב-`:app.js`:

```
import React from 'react';
import './App.css';
import MyContainer from './MyContainer';

function App() {

  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 מהיים. לכל קומפוננטה וקומפוננטה יש API מיוחד. כך, למשל, ב-UI 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 דפדף חדשה נעשית באמצעות:

```
window.open(props.linkTo);
```

פתרון:

```
import React from 'react';
import { makeStyles } from '@material-ui/core/styles';
import Card from '@material-ui/core/Card';
import CardActionArea from '@material-ui/core/CardActionArea';
import CardActions from '@material-ui/core/CardActions';
importCardContent from '@material-ui/core/CardContent';
import CardMedia from '@material-ui/core/CardMedia';
import Button from '@material-ui/core/Button';
import Typography from '@material-ui/core/Typography';

const useStyles = makeStyles({
  card: {
    width: 500,
  },
  media: {
    height: 140,
  },
});

export default (props) => {
  const classes = useStyles();

  function goTo() {
    window.open(props.linkTo);
  }
}
```

```

return (
  <Card className={classes.card}>
    <CardActionArea>
      <CardMedia
        className={classes.media}
        image={props.imageSrc}
        title="Random Image"
      />
      <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 onClick={goTo} size="small" color="primary">
        Learn More
      </Button>
    </CardActions>
  </Card>
);
}

```

אנו צריכים לזכור לא להתבלבל ולא לפחד משפע הkomponenot שיש בה. יש כפתור? מצוין, אפשר להעביר אליו onClick, בדיק כפ' של מדרנו. הכפתור, שהוא רכיב של UI, יודע בה

מואוד לטפל באירוע. אנו מעבירים לו אירוע פשוט שפותח מה שאנו מעבירים לו ב-props. במקרה זה:

```
function goTo() {
  window.open(props.linkTo);
}
```

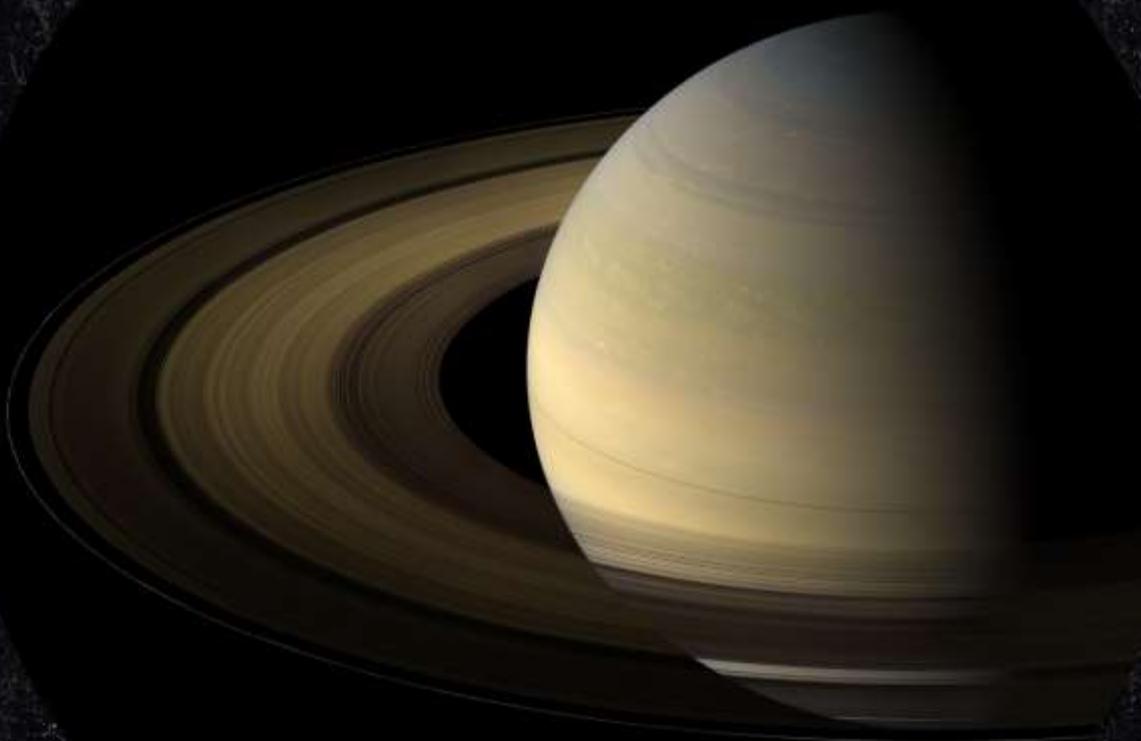
כל מה שעשינו לעשות הוא להעביר לקומפוננטה שלנו את הקישור. במקרה זה האדרט שהפתרנו יהיה So.linkTo.

```
<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>
```

כרגע אנחנו יכולים להשתמש ולפתח על בסיס הקומפוננטה. שימושו לב שימוש נטול, במקרה לחזור על קוד, להשתמש בפונקציית map.

פרק 18

HOC: HIGHER ORDER COMPONENT



HOC: Higher Order Component

מדובר בפיצ'ר של ריאקט שבאמצעותו אנו לוקחים קומפוננטה אחת ומעשירים אותה ביכולות נוספות ללא צורך להכיר את הקומפוננטה המוענשת. העשרה זו מתבצעת באמצעות פונקציה עוטפת שמקבלת כארוגמנט קומפוננטה. הפונקציה שמבצעת את השינוי נקראת **HOC**, ראשי תיבות של Higher Order Component. המשמעות של המושג זהה היא "קומפוננטות מדרגת גבוהה" – קלומר אליו שלא ממליכות את הידיים בפונקציונליות אלא מנפקות קומפוננטות אחרות, שהן אלו שעשוות את העבודה.

האמת היא שכמתכני ריאקט מתחילה לא יצא לכם לכתוב הרבה HOC, אך בוודאי יצא לכם להשתמש בהם והשימוש בהם עלול להיות מבלבל או סתום יכול להיראות כמו ודו – קלומר CIS שלא ברור איך הוא עובד.

כשאנו כותבים HOC, אנו צריכים לזכור שהקלט הוא קומפוננטה והפלט הוא קומפוננטה אחרת שימושת בהקומפוננטה של הקולט בסיסי. משחו בסגנון זהה:

```
import React from 'react';

const HighOrderComponent = (WrappedComponent) => {
  function HOC(){
    // Stuff
    return <WrappedComponent />;
  }
  return HOC;
};

export default HighOrderComponent;

const SimpleHOC = HighOrderComponent(MyComponent);
```

מה קורה בפונקציה זו? כיצד אני מעשיר את הקומפוננטה? הבה נדגים עם קומפוננטה שאני רוצה להעшир אך ורק בתוכנה אחת שאני רוצה להעביר לה. איזו? משמעות הח'ים, הק'ום וכל השאר. ערך התוכנה זו, כפי שידוע כל מי שקרא את הספר "מדריך הטרמפיקט לגלקסיה", הוא 42. איך אני מעשיר את הקומפוננטה שלי? אני עבורך אליה את כל ה-props שהיא מקבלת (אני לא יודע אילו) וגם את התשובה לח'ים עצם. בדוק:

```
import React from 'react';
```

```

const HighOrderComponent = (WrappedComponent) => {
  function HOC(props) {
    const answerToLifeMeaning = 42;
    return <WrappedComponent
      {...props}
      theAnswer={answerToLifeMeaning}
    />;
  }
  return HOC;
};

export default HighOrderComponent;

const SimpleHOC = HighOrderComponent(MyComponent);

```

הדבר שהכי מפחיד מתכנתים צעירים הוא ה-*the-spread operator*

```
{...props}
```

אבל מה שהוא עושה זה ללקח את כל ה-*props* שמעברים ולהעביר אותם, איך שהם, בפורמט מסודר, אל הקומפוננטה שאנו עוטף, בלי שאני אctrך להכניס אותם ידנית. בנוסף על כך, אני מעביר גם את משמעות החיים.

זו דוגמה מאוד תיאורטיבית, אבל היא מסיימת להבין עד כמה HOC הם פשוטים למרות הסינטקיס המאiev. פשוט לוקחים קומפוננטה אחת ומוסיפים לה תכונות או מתודות. הבה נראה דוגמה מעשית יותר. באחד הפרקים הקודמים יצרנו קומפוננטה שמכילה סטייט עם מספר, שאוначת ל-0, ולחיצה על הכפתור גורמת למונה לעלות ב-1. מה הבעיה? בכל פעם שביצענו רענון לדף, הסטייט התAES והמונה חוזר ל-1. הסטייט נשאר בזיכרון אבל רק לאורך חי' הקומפוננטה. עם HOC אנו יכולים להעשור כל קומפוננטה שהיא ולחשוף API שיבצע שמירה של המספר הזה.

הקומפוננטה נראה כר:

```

import React, { useState } from 'react';
import Button from '@material-ui/core/Button';

function CounterUp() {

  const [count, setCount] = useState(0);

```

```

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

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

export default CounterUp;

```

אתם יכולים ומוזמנים להציב אותה אצלכם ולראות שהיא עובדת באופן מושלם. אך הבעיה היא שברגע שאתם טוענים מחדש את הדף (באמצעות F5 למשל), המונה מתאפס וחוזר ל-0. אנו רוצים להעшир את הקומפוננטה הזו באמצעות HOC על מנת לשמור את המונה בזיכרון הדף.

שמירה ב"זיכרון" של הדף, באופן ישיר 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);
```

ראו לציין שאין שום קשר בין `localStorage` לריאקט. מדובר בפיצ'ר של הדף ושל הג'אווסקריפט שרצ עליו, בדיק כמו `fetch`, `fetch`, למשל, המשמש ל-AJAX בסביבת שרת.

fonkzit ha-HOC shel tikkach cil komponenta vutofig la matodot save v-load. lponkzit ani akra :WithStorage

```

import React from 'react';

const WithStorage = (WrappedComponent) => {
  function HOC(props) {

```

```

function save(key, data) {
  localStorage.setItem(key, data);
}

function load(key) {
  return localStorage.getItem(key);
}

return <WrappedComponent
  {...props}
  save={save}
  load={load}
/>;
}

return HOC;
};

export default WithStorage;

```

מה היא עשויה? מוסיפה לכל קומפוננטה שעוברת דרך שתி מתודות פשוטות: `save` ו-`load`. מתודה `save` שומרת ל-`localStorage` וממתודה `load` טעונה מה-`localStorage`. ואיך אני מעביר קומפוננטה כזו? אני פשוט זוכר שהקומפוננטה שעוברת דרך ה-HOC מקבלת ל-`props` שלה כמה תכונות חדשות, במקרה שלנו `save` ו-`load`.

ראשית, הטעינה עצמה נראה כך:

```

import React from 'react';
import WithStorage from './WithStorage';
import CounterUp from './CounterUp';

const ComposedComponent = WithStorage(CounterUp);

function MyContainer() {
  return (
    <ComposedComponent />
  );
}

```

```
export default MyContainer;
```

בutor קומפוננטת אב אני מבצע import לשתי הקומפוננטות שלו: ה-HOC והקומפוננטה שעוברת דרכו, וזה אני יוצר משתנה שאלוי אני מוחזר את הקומפוננטה המועשת:

```
const ComposedComponent = WithStorage(CounterUp);
```

בקומפוננטה הזאת אני יכול להשתמש כמו בכל קומפוננטה אחרת. צריך לזכור שה-HOC, במקרה זה WithStorage, תמיד מוחזר לי קומפוננטה שאני יכול להשתמש בה ב- JSX. זה בדיק מה שאני עושה בקומפוננטת ה-קונטינר:

```
function MyContainer() {
  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 `:save`:

```
import React, { useState, useEffect } from 'react';
import Button from '@material-ui/core/Button';

function CounterUp(props) {
  const [count, setCount] = useState(0);

  function increaseHandler() {
    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 שמוופעל לאחר לחיצה על הכפתור, אני קורא ל-props.save ומעביר דרכו שני פרמטרים: ה-key שהחלהטי 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 שלו הוא מערך ריק. ב-`useEffect`, שקורה רק כאשר הקומפוננטה מתעדנת לראשונה, אני אקרא למתודת `load` עם המפתח (שהוא `counter`), אמיר אותו למספר (כל הערכים ב-`localStorage` נשמרים כמחרוזת טקסט) ואכניס אותו לסטיטו. אם אין כלום ב-`localStorage`, אני מחזיר מספר.

```
import React, { useState, useEffect } from 'react';
import Button from '@material-ui/core/Button';

function CounterUp(props) {
  useEffect(() => {
    let counter;
    if(props.load) {
      counter = props.load('counter');
    }
    const initialCounter = Number(counter) || 0;
    setCount(initialCounter);
  }, []);
  const [count, setCount] = useState(0);

  function increaseHandler() {
    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 לבלבול. הנה ננתח אותו שוב. מדובר בפונקציה שמקבלת שני פרמטרים: פונקציה אונימית ומערך ריק. המערך הריק נועד להורות לריאקט להפעיל את הפונקציה האונימית רק פעם אחת, בתחילת טעינת הקומפוננטה.

בפונקציה האנומלית יש קרייה ל-`load.loadProps`, המתוודה שניתנה לנו על ידי ה-HOC. אם היא נמצאת אני ממיר אותה למספר. אם לא, אני מחריר 0 והcoil נכנס לסת"ט. זה הכל. אני בודק, כמובן, אם `props.load` קיימת.

از למה להכנס לכל הסרט הזה של HOC ולא פשוט להכניס את `localStorage.setItem` ו-

ה-`localStorage.getItem` לקומפוננטת CounterUp מההתחלה? למה לטרוח לעטוף אותה ב-HOC? התשובה היא שעכשיו אני יכול לחת את ה-HOC הזה ולעתוף אליו קומפוננטות נוספות ולתת להן, כמובן, את היכולת לקרוא ולכתוב לתוך ה-`localStorage` שלו בלי להתאים כלל ובלי לכתוב שוב ושוב קסם, את היכולת לקרוא ולכתוב לתוך ה-`localStorage` שלו בלא כדי לפגוע בשמונת המונחים קוד שאנחנו צריכים להטמע בקומפוננטה שלנו ברגע לתקשרות לצד השרת זהה וחושך המונח-המן עוד מיותר. הקראות לצד השרת, או במקרה הזה ל-`localStorage`, תמיד יהיו מרכזיות במקום אחד.

תרגיל:

הוסיף ל-HOC שכתבנו `WithStorage` – Method `clear` שמנקה את ה-`localStorage`.
רמז: ניקוי ה-`localStorage` נעשה באמצעות:

```
localStorage.clear();
```

פתרון:

```
import React from 'react';

const WithStorage = (WrappedComponent) => {
  function HOC(props) {
    function save(key, data) {
      localStorage.setItem(key, data);
    }
    function load(key) {
      return localStorage.getItem(key);
    }
    function clear() {
      localStorage.clear();
    }
    return <WrappedComponent
      {...props}
      save={save}
      Load={load}
      clear={clear}
    />;
  }
  return HOC;
};

export default WithStorage;
```

אין בעיה להוסיף כמה פונקציות שנרצה ל-HOC ולהעшир בכך שבא לנו את הקומפוננטות שעובדות דרך ה-HOC. במקרה זהה פשוט הוספנו מתודה נוספת `clear` שנקראת `clear` ועושה את הפעולה הבאה:

```
localStorage.clear();
```

תרגילים:

בקומפוננט CounterUp שהודגמה בפרק ועטופה ב-HOC WithStorage בשם CounterUp, הוסיף כפטור שմבצע איפוס של המונה.

רמז: איפוס המונה מתבצע באמצעות הפעלת מתודה clear שהוספנו אל WithStorage HOC. יש ליצור כפטור שיפעל את המתודה זו.

פתרונות:

```
import React, { useState, useEffect } from 'react';
import Button from '@material-ui/core/Button';

function CounterUp(props) {
  useEffect(() => {
    const initialCounter = Number(props.load('counter')) || 0;
    setCount(initialCounter);
  }, []);

  const [count, setCount] = useState(0);

  function increaseHandler() {
    setCount(count + 1);
    if (props.save) {
      props.save('counter', count + 1);
    }
  }

  function resetHandler() {
    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 להוציא כפתור. הכפתור הוא כפתור ה-`:reset`

```

<Button variant="contained" color="primary"
onClick={resetHandler}>Reset</Button>

```

ב-`onClick` שלו מופעלת הפונקציה `props.clear`. היא תפעיל בטורה את ה-`resetHandler` וגם תאפשר את הסטייט:

```

function resetHandler() {
  setCount(0);
  if (props.clear) {
    props.clear();
  }
}

```

מайפה ה-HOC מפעיל props.clear שנותף את הקומפוננטה. זו הסיבה שאנו בודקים אם הוא קיים. איפה הוא עותף את הקומפוננטה הזאת? הראיינו בפרק. בקומפוננטה Container:

```
import React from 'react';
import WithStorage from './WithStorage';
import CounterUp from './CounterUp';

const ComposedComponent = WithStorage(CounterUp);

function MyContainer() {
  return (
    <ComposedComponent />
  );
}

export default MyContainer;
```

פרק 19

לאוֹטִינְגֶּר



ראוטינג

עד כה עבדנו עם אפליקציות בנettes דף אחד שבו יש קומפוננטות שונות. אבל זה כמובן לא ריאלי. באתר אמיתי, או באפליקציה אמיתית, יש כמה וכמה דפים. באתר אינטראנט בסגנון הישן, כל מעבר דף גרם לטעינה מחודשת של האתר כולו. בריект (וכן בכל אפליקציות הוווב המודרניות) מעבר דפים נועשים ללא טעינה כלל מהשרת ובאמצעות קומפוננטות ריאוטינג שמבצעות ניתוב של קומפוננטות דרך צד הליקוח, דבר המאפשר חוויה גלית טוביה בהרבה. הטכניקה זו נקראת SPA – ראשי תיבות של Single Page Application.

ראוטינג (Routing), או בעברית תקנית "ניתוב", הוא מונח ידוע בכלל הנוגע לאפליקציות וbsites, ומובן שיש אותו גם בריект. ריאקט, ברגע לפירימורקים אחרים (כמו אングולר), לא קובעת איך למש את הריאוטינג, אלא יש כמה קומפוננטות מסוימות 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 במאורות ובאלפי אתרים אחרים.

האובייקט `>window.location` המכיל את הכתובת נקרא `location`. אפשר למצוא אותו תחת `window.history` ונתן יכולות לשנות אותו באמצעות `location.replace()`.

הבעיה היא ששינוי שלו גורם לטעינה מחדש, כפי שכבר למדנו, גורם לרנדור מחדש של כל הקומפוננטות ויוצר חווית משתמש לא טוביה, וגם עשוי לגרום בעית ביצועים. הפתרון? לשנות את ה-`location` בלי לטען את הדף מחדש. יש שתי דרכים לעשות זאת זהה: באמצעות שימוש באובייקט `History`, שנמצא תחת `window.history`, או באמצעות תוספת לכתובת בשם `hash`. השימוש ב-`hash` נחשב למתќדם יותר והשימוש ב-`hash` נחשב פשוט יותר. הנה נציגים ריאוטינג באמצעות `hash` בלבד.

חשוב: זה הסימן `#` שנמצא ב-URL. למשל: `example.com#page1`

השרת אינו מסוגל "לראות" את מה שיש מאחורי הסימן `#` וכשאנו מושנים אותו בדף, הוא לא נשלח כלל לשרת. זו הסיבה ששינויו שלו, אפילו ידנית, לא יטعن מחדש את הדף. בגלל זה `#` פופולרי מאד לריאוטינג ב-`hash` כ-`Single Page Application` כמו באפליקציות ריאקט.

על מנת לתרגם ריאוטינג ניזור לשולש קומפוננטות פשוטות שניבור ביבנה. `Home`, `About` ו-`Help`. בדוגמה הבא אני מכניס בהן רק כתורת פשוטה, אבל הן יכולות להכיל דפים שלמים וקומפוננטות אחרות, כמובן.

קומפוננטת `Hash.js`:

```
import React from 'react';

function Home() {
  return (
    <div>
      <h1>Hello World</h1>
      <p>This is the Home component</p>
    </div>
  );
}

export default Home;
```

```
<div><h1>Home</h1></div>
);
}
```

```
export default Home;
```

קומponent **xsx**

```
import React from 'react';
```

```
function About() {
```

```
    return (
```

```
<div><h1>About</h1></div>
```

```
);
}
```

```
}
```

```
export default About;
```

קומponeנטת Help.js:

```
import React from 'react';

function Help() {
  return (
    <div><h1>Help</h1></div>
  );
}

export default Help;
```

כמובן, אלו קומponeנטות פשוטות לשם לימוד הריאקטינג והבנתו בלבד. כדי לבצע ריאקטינג אנו יוצרים קומponeנטת אב שמכילה את סרגל הניווט ואת הקומponeנטה שת��ען בכל פעם את קומponeנטת העמוד הפעילה, כמו About, Home, או את Help.

קומponeנטת האב בעצם נתלית באירוע בשם hashchange שנמצא על window. האירוע זה מתרחש בכל פעם שימושו משנה את הכתובת שיש אחרי #-#. כך, למשל, אם אני נמצא בדף example.com#page1 – האירוע יופעל כתוצאה של קישור שמוביל 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 הזה page2#.

`substr(1)`

החלק השני מוציא את #-# ומותיר אותו עם page2.

נבין את זרימת המידע בקומפוננטה לפני הכתיבה שלה:

1. בתחילת הקומפוננטה מחברת לאיורע `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. מכניסים לסתיט את ה-`route` הפעיל.
4. השני בסטייט גורם לקומפוננטה להתרנדר מחדש. ברנדור בודקים את ה-`route` הפעיל ובחרים את הקומפוננטה.

אחרי שהבנו את הצעדים ההכרחיים – נצפה בקומפוננטה המאפשרת את זה:

```
import React, { useState, useEffect } from 'react';
import Home from './Home';
import Help from './Help';
import About from './About';

function Router() {

  const [route, setRoute] =
  useState(window.location.hash.substr(1));

  useEffect(() => {
    window.addEventListener('hashchange', () => {
      setRoute(window.location.hash.substr(1));
    })
  }, []);

  let Child;

  function getChild() {
    switch (route) {
      case '/about':
        Child = About;
        break;
      case '/help':
        Child = Help;
        break;
      default:
        Child = Home;
    }
  }

  return (
    <div>
      {getChild()}
      <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 נציב כמובן ב-`App.js` באפליקציה שלנו:

```

import React from 'react';
import './App.css';
import Router from './Router';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Router />
      </header>
    </div>
  );
}

export default App;

```

נסזה להבין מה קורה בקומפוננטת `Router`. הדבר הראשון שאינו עושה הוא ליצור סטייט בשם `route` ולתת לו ערך ראשון של מה שיש בכתבota #:

```
const [route, setRoute] = useState(window.location.hash.substr(1));
```

למשל, אם אני טוען את הקומפוננטה כשאני נמצא ב:

<http://localhost:3000/#/home>

הסטייט יהיה `/home`. הקוד יחזיר לי את השורה הבאה:

```
useState(window.location.hash.substr(1))
```

הדבר השני שאני עושה הוא שאני גורם לקומפוננטה, באמצעות הhook `useEffect`, בrendsור הראשוני בלבד, להצמיד אירוע שישנה את ה-`route` בסטייט בהתאם למה שיש אחרי #-#. אני מודא שמדובר בrendsור הראשוני בלבד כי אני מעביר ב-`useEffect` פרמטר שני של מערך ריק:

```
useEffect(() => {
  window.addEventListener('hashchange', () => {
    setRoute(window.location.hash.substr(1));
  })
}, []);
```

שימוש לב: מפני שהפרמטר השני שמועבר ל-`useEffect` הוא מערך ריק [], אז הפונקציה שאנו מעבירים תופעל רק בrendsור הראשוני של הקומפוננטה – כלומר פעם אחת. זה חשוב, כי אם נצמיד אירוע ל-`hashchange` בכל בrendsור של הקומפוננטה זו תהיה לנו זילגת זיכרון חמורה. אנו מצמידים אירוע פעם אחת בלבד.

הדבר השלישי הוא ליצור פונקציה פשוטה שעושה בדיקה מה הסטייט הפעיל ומחליפה את משתנה Child בקומפוננטה המתאימה לסטטוס. שימו לב שאנו משתמש במשתנה שמתחליל באות גדולות – `Child` ולא `child`. זה משומם שמדובר במשתנה המכיל קומפוננטה ואני מציין בכך ריאקט שמדובר בקומפוננטה:

```
let Child;

function getChild() {
  switch (route) {
    case '/about':
      Child = About;
      break;
    case '/help':
      Child = Help;
      break;
    default:
      Child = Home;
  }
}
```

חשוב: כדאי לשים לב שהמשתנים `Home`, `About`, `Help` מגעים מהצחרת `the`-`import`.

הצעד האחרון הוא לрендר את התוצאה:

```
return (
  <div>
    {getChild()}
    <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. אני מציב את getChild לפני הקריאה לקומפוננטה עצמה. כלומר המשתמש לוחץ על קישור, והאיירוע של hashchange עובד ומשנה את הסטייט. שניי הסטייט גורם לרנדור מחדש, שבו Child מקבל קומפוננטה חדשה.

כאמור, כך זה נראה באופן מאוד גס ופשוט. אם נרצה שזה יעבד עם קישורים אמיתיים ולא עם hash נצטרך להשתמש ברכיבים שונים במקצת ולהפריד, כמובן, בין הקונפיגורציה לרכיב הראותינו (למשל לשים את כל הנתיבים בקובץ JSON או על השרת), אבל בגודל – כך זה נראה. ראותינו בראקט איננו וודו וקל לממש אותו, אבל בעולם האמיתי נעדיף להשתמש בקומפוננטה מוכנה מראש.

הקומפוננטה הפופולרית ביותר נקראת, בפשטות, react-router, ואני נלמד עליה בנוגע לראוטינג. אוטם העקרונות המנחים אותנו רלוונטיים גם לראוטינג של כל קומפוננטה. הקומפוננטה הזאת משתמשת ממש בכתובות אמיתיות ולא #-#, אך העיקרון זהה. react-router היא ספריית ראותינו כללית וספרייה נוספת שלה, react-router-dom, היא המשילמה לאפליקציות ווב. אם כך, נלמד להשתמש בשתייהן.

ראשית, ניכנו לדף של react-router

<https://github.com/ReactTraining/react-router>

אפשר לראות שיש שם מדריך מסודר וdockumentציה טובה. זה כלל מפתח בוגע לכל קומפוננטה שאתה רוצה להשתמש בה בראקט (ובכל מערכת) – dockumentציה טובה היא הכרחית. אם אין dockumentציה, עדיף לא להשתמש באוותה קומפוננטה. אבל זה לא המקרה פה.

כיוון שמדובר בקומפוננטה חיונית, אנו נתקין אותה. ההתקנה נעשית, בדיק כמו כל קומפוננטה חיונית אחרת, באמצעות וקח שעליו למדנו בקורס בפרק על קומפוננטות חיוניות. נכנסים עם react-router-al המיקום של הפרויקט שלנו ומתקינים את komponentet react-router ואת dom באמצעות:

```
npm install react-router-dom
```

גם כאן, על מנת להדגים, משתמש בkomponentot Home, Help>About. בנויג לדוגמה שהראינו, של react-router-dom עובדת עם כתובות אינטרנט אמיטיות – קלומר ללא הסימן #. יש לך כמה יתרונות והמיושן געשה באוותה הדרך.

המתודולוגיה זהה. אני מגדיר את הראוטינג ואיזו קומפוננטה נטענת בעקבותיו:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom';
import Home from './Home';
import Help from './Help';
import About from './About';

function MyRouter() {
  return (
    <Router>
      <div>
        <h1>App</h1>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/help">Help</Link>
          </li>
        </ul>
        <hr />
      <Switch>
        <Route exact path="/">
          <Home />
```

```

        </Route>
        <Route path="/about">
            <About />
        </Route>
        <Route path="/help">
            <Help />
        </Route>
    </Switch>
</div>
</Router>
);
}

export default MyRouter;

```

אם עברתם על תת-הפרק שמסביר איך מבצעים את הרואTING ללא מודול חיצוני, זה יכול להיראות לכם פשוט עד כדי גיחוך. מאחורי הקומפוננטה יש שתי תפיסות עיקריות:

1. כל קישור לנútב נעשה באמצעות [קומפוננטת Link](#).
2. הקומפוננטה הראשית היא Router וה-Switch ממומש לא ב-Switch רגיל אלא בעזרה קומפוננטה.

זה הכל. ברגע שambilים איך Router עובד באופן טבעי, גם הקומפוננטות של Router נראות פשוטות.

כל מאד להשתמש גם בפרמטרים של URL. בדוקומנטציה מוסבר איך לעשות זאת. כל מה שעשינו:
לעתות הוא להויסף את הפרמטר עם נקודותים בקומפוננטת Route שנמצאת בתוך ה-Switch:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom';
import Home from './Home';
import Help from './Help';
import About from './About';

function MyRouter() {
  return (
    <Router>
      <div>
        <h1>App</h1>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/help/6382020">Help</Link>
          </li>
        </ul>
        <hr />
      <Switch>
```

```

<Route exact path="/">
  <Home />
</Route>
<Route path="/about">
  <About />
</Route>
<Route path="/help/:id">
  <Help />
</Route>
</Switch>
</div>
</Router>
);
}

export default MyRouter;

```

השינוי היחיד שבוצע מהרארוטינג הבסיסי יותר הוא:

<Route path="/help/:id">
זה בעצם מאפשר לנו להכניס פרמטרים, למשל:

<http://localhost:3000/help/6382020>

הפרמטר זה יהיה זמין לנו בקומפוננטת Help באמצעות `useParams`, שהיא פונקציה שmag'ua עם `.react-router`. למשל כך:

```
import React from 'react';
import {
  useParams
} from 'react-router-dom';

function Help() {
  let { id } = useParams();
  return (
    <div><h1>Help. The ID is {id}</h1></div>
  );
}

export default Help;
```

נסו את זה בעצמכם. הציבו את הקוד הזה ב-Create React App שלכם וראו עד כמה זה פשוט. בדיקומנטציה של `react-router` יש דוגמאות רבות לשימוש שכדי לבחון. אבל כאמור, אם הבנתם איך ממשים ריאוטינג בעצמכם ואייר העיקרון עובד, לא יהיה לכם קשה להבין ולהשתמש בקומפוננטות של ניווט.

תרגיל:

צרו שתי קומפוננטות: `Home`, שזהה לקומפוננטה מתחילה הפרק, והקומפוננטה הזו, המציגה תמונה אקראיית מהרשת:

```
import React from 'react';

function NiceImage() {
  let imgSrc = 'https://picsum.photos/id/237/200/300';
  return (
    <img src={imgSrc} />
  );
}

export default NiceImage;
```

צרו אפליקציית ריאקט שמשתמש בראוטינג. באמצעות הרואוטינג אפשר להחליף בין `Home` לבין `NiceImage`. אל תשתמשו בראוטינג של קומפוננטה חיצונית.

פתרון:

```
import React, { useState, useEffect } from 'react';
import Home from './Home';
import NiceImage from './NiceImage';

function Router() {

  const [route, setRoute] =
  useState(window.location.hash.substr(1));

  useEffect(() => {
    window.addEventListener('hashchange', () => {
      setRoute(window.location.hash.substr(1));
    })
  }, []);

  let Child;
```

```

function getChild() {
  switch (route) {
    case '/image':
      Child = NiceImage;
      break;
    default:
      Child = Home;
  }
}

return (
  <div>
    {getChild()}
    <h1>App</h1>
    <ul>
      <li><a href="#/image">NiceImage</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`, שਮופעל בכל פעם שבה-`hash` (שזה #-ב-URL משתנה, מחליף את משתנה ה-`route` בהתאם לערך של מה שיש אחריו ה-`hash` וטוען קומפוננטה אחרת).

תרגילים:

ממשו את הפתרון הקודם באמצעות `.react-router`

פתרונות:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom';
import Home from './Home';
import NiceImage from './NiceImage';

function MyRouter() {
  return (
    <Router>
      <div>
        <h1>App</h1>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/image">Nice Image</Link>
          </li>
        </ul>
        <Switch>
          <Route exact path="/">
            <Home />
          </Route>
          <Route path="/image">
```

```

    <NiceImage />
  </Route>
  </Switch>
</div>
</Router>
);
}

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. עטיפת כל הקומפוננטה בקומפוננטה ראשית מסווג Router.
 5. תפריט וקישורים באמצעות רכיב Link: <Link to="/image">Nice Image</Link>
 6. יצירת קומפוננטת Switch במקום אמרות להיות הקומפוננטות שאני מגיע אליה בינויו.
 7. להציב בתוך Switch את הקומפוננטות שנטענות כתוצאה מפעולות הראוטינג.
- למשל: <Route path="/image"><NiceImage /></Route>

תרגילים:

בפתרון התרגיל הקודם, אפשר להחליף את התמונה באמצעות שינוי מספר ה-`id` באופן הבא:

`https://picsum.photos/id/${id}/200/300`

כלומר ה-`id` יכול להיות:

`https://picsum.photos/id/100/200/300`

או:

`https://picsum.photos/id/44/200/300`

וכך הלאה.

הכניםו ניוט באמצעות פרמטרים, כך שתוכלו להכניס מספר ב-URL, למשל:

`http://localhost:3000/image/100`

או:

`http://localhost:3000/image/44`

והתמונה עם ה-`id` המתאים תיתען.

פתרונות:

קומפוננטת הראותינו:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom';
import Home from './Home';
import NiceImage from './NiceImage';

function MyRouter() {
  return (
    <Router>
      <div>
        <h1>App</h1>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/image/1">Nice Image</Link>
          </li>
        </ul>
        <Switch>
          <Route exact path="/">
            <Home />
          </Route>
          <Route path="/image/:id">
            <NiceImage />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}
```

```

    </Switch>
  </div>
</Router>
);
}

export default MyRouter;

```

:NiceImage קומפוננטה

```

import React from 'react';
import {
  useParams
} from 'react-router-dom';

function NiceImage() {
  let { id } = useParams();

  let imgSrc = `https://picsum.photos/id/${id}/200/300`;
  return (
    <img src={imgSrc} />
  );
}

export default NiceImage;

```

בראוטינג הפתרון הוא פשוט יותר – הוספת הפרמטר `id` ב-`path` בקומפוננטת `Route`.
 בקומפוננטה שאמורה ל取פוא את `id`, הלא היא `NiceImage`, צריך לבצע import ל-`useParams`.
 שmag'ua אלינו מ-`react-router`, קיבל את `id` וילשוח לו `src` ב-`img`.
 אפשר ממש גם שדה או תיבת `Select` שבום כתבים מספר, וההתמונה המתאימה תיתען.

פרק 20

הונתקם



קונטקסט

קונטקסט הוא מונח חשוב בראקט. הוא מאפשר לנו לחלק מידע בין קומפוננטות שונות, בדומה לסטיטיט, אבל הוא גלובלי. כו"ם אנו מכירים שתי דרכיים להעברת מידע בקומפוננטות: האחת היא באמצעות `props`, כאשר קומפוננטת האב מעבירה לקומפוננטת הבנות מידע דרך ה-`props`, והשנייה היא אירועים – קומפוננטת הבת מפעילה אירוע כתוצאה מפעולה כלשהי (משתמש או טימר). אם קומפוננטת האב העבירה פונקציה שモפעלת כתוצאה אירוע כלשהו (`Handler`), הוא יופעל כאשר אירוע מופעל בקומפוננטת הבת.

הבעיה היא שזה יכול להסתבר. לעיתים קומפוננטות רבות מעכיז 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 להלהה. לעיתים יש מידע בקומפוננטה אב מסוימת וצריך להעביר אותו לקומפוננטה שנמצאת חמיש רמות מתחתיה בהיררכיה ולשרשר את המידע. זה מורכב. קונטקסט מאפשר לנו לשמור מגיר מידע גדול שמננו כל קומפוננטה יכולה לקרוא, מעין סטייט גלובלי.

נכון לארסה האחורה של ריאקט, מומלץ להשתמש בקונטקסט אך ורק למיידע שלא מתעדכן באופן תכוף. לעדכנים באופן תכוף מומלץ להשתמש ברידקס, שבו נdon באחד הפרקים הבאים, או באלטרנטיבות אחרות לניהול סטייט גלובלי כמו `mobx` או `observables`, שאין נלמודות בספר זה.

בקונטקסט יש שני משתתפים: האחד הוא הפרוביידר (`Provider`) והשני הוא הצרךן (`Consumer`). הפרוביידר חייב תמיד לעטוף את הצרךן ולהיות גבוהה יותר ממנו בהיררכיה, אחרת הצרךן לא יוכל לגשת לקונטקסט שאותו הפרוביידר מספק. הפרוביידר יכול לאכלס את הקונטקסט באופן סטטי או לבצע קריאה ל-`API` חיצוני. הצרךן מקבל גישה לקונטקסט באמצעות הוקים.

פרוביידר לא חייב להיות קומפוננטה ובדרך כלל לא בניי קומפוננטה אלא קובץ ג'אווהסקריפט רגיל, למשל `UserContext.js`, שמאגדיר קונטקסט של משתמש:

```
import React from 'react'

const UserContext = React.createContext({});

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({});
```

האובייקט הריק, שמעבר כפרמטר ל-`createContext`, הוא בעצם בירית מיחל – ערך שמוחזר רק כאשר קומפוננטה מבצעת קרייה לפרוביידר זהה, אם הוא לא משוייך אליה. זה אידיאלי לבדיקות, אבל כרגע תשאירו אותו כאובייקט ריק.

אני משתמש ב-API של קונטקסט כדי ליצור וליצא את הפרוביידר, את הערך ואת הקונטקסט. אני יכול לבחור לפרוביידר, לצריך או לקונטקסט תחילה כרצוני. במקרה הזה, כיון שמדובר בקונטקסט של משתמש, בחרתי בתחילת של `User`.

אחרי שהגדרתי קונטקסט, אני יכול להשתמש בפרוביידר. הפרוביידר מספק לכל הקומפוננטות שutztפות אותו את הקונטקסט שהוא נוטן. אני יכול לשימושו בכל מקום, אפילו ב-`App`. הנה דוגמה:

```
import React from 'react';
import './App.css';
import { UserProvider } from './UserContext'

function App() {

  const user = {
    name: 'Ran',
    surName: 'Bar-Zik',
    city: 'Petah-Tiqwa'
  };

  return (
    <div className="App">
      <header className="App-header">
        <UserProvider value={user}>
          Any component
        </UserProvider>
      </header>
    </div>
  );
}

export default App;
```

ראשית אני מיבא את הפרוביידר מהקונטקסט שיצרתתי. כזכור, אני מיצא שם פרוביידר, צריך וקונטקסט.

```
import { UserProvider } from './UserContext'
```

אני בוחר את המידע שאינו רוצה להעביר בקונטקסט. המידע יכול להיות סטטי או מידע ש מגיע משירות (כolumbia mahashet). בדוגמה זו אני משתמש במידע סטטי:

```
const user = {
  name: 'Ran',
  surName: 'Bar-Zik',
  city: 'Petah-Tiqwa'
};
```

הצעד האחרון הוא להשתמש בפרוביידר כמו בקומפוננטה רגילה ולהעביר אליו את המידע. כל מה שיהיה עטוף בקומפוננטה זו וgemäßן כל הקומפוננטות שמתוחת לקומפוננטה העטופה, כולמר אלו שתחתיתיה בהיררכיה, יקבלו גישה לكونטקסט:

```
<UserProvider value={user}>
  Any component
</UserProvider>
```

עכשו במקום `Welcome.js` נשים קומפוננטה אמיתית, למשל `Any Component`:

```
<UserProvider value={user}>
  <Welcome />
</UserProvider>
```

לא נשכח לעשות `import`, כמובן, לפני שאנו מציבים אותה.

air הקומפוננטה זו תיראה?

```
import React, { useContext } from 'react'
import UserContext from './UserContext'

function Welcome() {
  const user = useContext(UserContext);

  return (
    <span>Hello, {user.name} {user.surName}</span>
  );
}

export default Welcome;
```

הקומפוננטה זו משתמשת בקונטקסט באמצעות הוק `useContext`, השלב הראשון הוא להביא הוק זה שעבוד בדיקן כמו כל הוק אחר:

```
import React, { useContext } from 'react'
```

השלב השני הוא להביא את הקונטקסט שהוא מעוניין להשתמש בו. במקרה זה, `UserContext`:

```
import UserContext from './UserContext'
```

השלב הבא והאחרון הוא פשוט לזרוך את המידע שהפרוביידר זהה מביא לנו, באמצעות:

```
const user = useContext(UserContext);
```

מהשלב זהה, כל מה שיש ב-`user` הוא מה שיש לנו בפרוביידר.

אם הפרויקט מודע למשתמש – הוא יrndר מיד את הקומפוננטות שמשתמשות בו. שימושו לבשהקומפוננטה צריכה להיות מודעת ל-`UserContext` ולו `import`.

אבל מה קורה אם אנו רוצים לעדכן את הקונטקסט ממוקם אחר שהוא לא הפרויקט? אם אנו רוצים לחת לפרויקט אפשרות לה汰עדן על ידי אחת מהקומפוננטות בתחתייה ההיררכיה, אנו צריכים להבהיר, נוסף על הקונטקסט, גם פונקציה שתשנה אותו.

אני אדגים זאת בעזרת קומפוננטת `Container`. קומפוננטת `Container` ממסת פרויקט שיש לו ערך שמאפשר מהסיטי וגם פונקציה שיכולה לשנות את הסיטי:

```
import React, { useState } from 'react';
import { UserProvider } from './UserContext'
import Welcome from './Welcome';

function Container() {

  const [user, setUser] = useState({
    name: 'Ran',
    surName: 'Bar-Zik',
    city: 'Petah-Tiqwa'
  });

  const providerOptions = {
    data: user,
    changeUser: (value) => setUser(value),
  }

  return (
    <div>
      <UserProvider value={providerOptions}>
        <Welcome />
      </UserProvider>
    </div>
  );
}

export default Container;
```

אפשר לראות שבמוקם להכניס רק את ערך אובייקט המשתמש, אני מכניס את ערך אובייקט המשתמש בסיטי ומעביר אובייקט שיש בו גם את הסיטי וגם פונקציה שמשנה אותו.

אם JSX רוצה לשנות את המידע הזה, הוא צריך להפעיל את הפונקציה שמשנה את הסטייט שבו נמצא הפלרוביידר, וכך הוא ישנה אותו:

```
import React, { useContext } from 'react'
import UserContext from './UserContext'
import Button from '@material-ui/core/Button';

function Welcome() {
  const user = useContext(UserContext).data;
  const changeUser = useContext(UserContext).changeUser;

  const newUser = {
    name: 'Moshe',
    surName: 'Cohen',
    city: 'Bat-Yam'
  }

  function clickHandler() {
    changeUser(newUser);
  }

  return (
    <div>
      <span>Hello, {user.name} {user.surName}!</span>
      <Button variant="contained" color="primary"
onClick={clickHandler}>Load another user</Button>
    </div>
  );
}

export default Welcome;
```

בקומפוננטת `Welcome`, ראשית אני שואב את אובייקט המשתמש (כדי שאוכל להציג אותו) וכן את הפונקציה שמשנה את הסטייט של הפלרוביידר. את הפונקציה אני מפעיל באירוע הלחיצה של הכפתור.

از בעצם מה שקרה הוא:

1. אני מגדיר קונטקסט בקובץ נפרד.
2. בוחר היכן להציב את הפרוביידר שלי, בדרך כלל בקומפוננטת אב. במקום שבו אני מציב את הפרוביידר, אני מזמין ערך. הדרך הכח טובה לעשות זאת היא באמצעות סטייט, אם כי זו לא חובה.
3. אני יכול להבהיר ייחד עם הערך פונקציה שמשנה את הסטייט של הפרוביידר. כל מי משתמש בקונטקסט זהה יכול (אם הוא רוצה) להשתמש בה ולשנות את הקונטקסט לכולם.

קונטקסט הוא דרך מצוינת לחלק מידע בין קומפוננטות שונות וגם קל להשתמש בו עם הווקים. הדרך לעדכן קונטקסט יכולה להיות מסורבלת, אבל אם מבינים שברגע שיוצרים קונטקסט ומידע גם יוצרים פונקציה שמשנה אותו ומעבירים את שניהם — אז הכל יהיה פשוט.

תרגילים:

העתיקו את קומפוננטת Welcome.jsx וקומפוננטת Container.jsx שלכם. השימו לב שה-`index.js` מכיל את ה-`Container.jsx` באוף זהה:

```
import React from 'react';
import './App.css';
import Container from './Container';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Container />
      </header>
    </div>
  );
}

export default App;
```

הוסיף קומפוננטת בת לكونטינר קומפוננטה בשם `CityName.jsx` שמציגה את שם העיר של המשתמש.

פתרונות:

ראשית, ניצור את הקומפוננטה. הקומפוננטה היא קומפוננטה שאמורה לצורך קונטקסט ונראית כך:

```
import React, { useContext } from 'react'
import UserContext from './UserContext'

function CityName() {
  const user = useContext(UserContext).data;

  return (
    <div>
      <span>You are from {user.city} </span>
    </div>
  );
}

export default CityName;
```

השורות המעניינות מבeginning הן:

```
import React, { useContext } from 'react'
const user = useContext(UserContext).data;
```

cut ניבא את הhook :useContext

כיוון שאנחנו יודעים שהקונטקסט מגיע כאובייקט שתכמה אחת שלו היא המידע ותכמה נוספת היא הפונקציה שמעדכנת אותו, אנו מעוניינים רק במידע ואנו מכניסים את ה-data אל המשתנה user. מפה מדובר בהדפסה פשוטה.

ומה קורה בקומפוננטת `cityName` ? אין חדש. אנחנו רק צריכים להציב את קומפוננטת `asj` .
קומפוננטה מתחת לפروب'ידר:

```
import React, { useState } from 'react';
import { UserProvider } from './UserContext'
import Welcome from './Welcome';
import CityName from './CityName';

function Container() {

  const [user, setUser] = useState({
    name: 'Ran',
    surName: 'Bar-Zik',
    city: 'Petah-Tiqwa'
  });

  const providerOptions = {
    data: user,
    changeUser: (value) => setUser(value),
  }

  return (
    <div>
      <UserProvider value={providerOptions}>
        <Welcome />
        <CityName />
      </UserProvider>
    </div>
  );
}

export default Container;
```

אפשר גם לשים את הקומפוננטה כקומפוננטה בת של `asj` . `Welcome` . כל עוד היא בת, נצדה, נינה או
הבת של הנינה של `UserProvider` זה לא משנה – היא יכולה להשתמש בקונטקסט.

פרק 21

חיבור לשורת נס סרווייסים



חיבור לשרת עם שירותי

פרק זה מניח ידוע מוקדם ב-XAJA ובהבדל בין הבקשות השונות – לפחות ברמת פונקציית fetch הקיימת.

חיבור לצד שירות הוא קרייטי בכל הנוגע לאתר או לאפליקציה המבוססים על ריאקט (או בכלל). רוב האתרים אינם אטרים סטטיים אלא מתעדכנים במועד המגיע מהשרת. המידיע המגיע מהשרת נכנס כבר כ-props לקומפוננטות מקומוננטות אב כלשהן או לكونטיקט. את הקוריאות אלו מבצעים באמצעות XAJA ואפשר להשתמש ב-fetch שmagiu עם ג'אוועסקריפט או בכל ספרייה אחרת, כמו הספרייה הנפוצה Axios.

כשאנו מרכיבים קוד בריект, הוא רץ בדף בלבד. נכון, App מרימה שירות פיתוח במחשב שלנו לשם הנוחות. אבל בסופו של יומם, כתוצאה הפרודקסן, הסביבה האמיתית, השירות יכול להיות שונה לחלוטין. הדף יכול לקבל את קובץ הג'אוועסקריפט שמורכבים מספריית ריאקט ומהקוד שלכם מכל שירות שהוא. אחרי שדף הלקו מורה את קובץ הג'אוועסקריפט האלון, הוא מרים את האפליקציה הבנוייה על ידי ריאקט על הדף בלבד. אם הוא צריך מידע מהשרת, הקוד שלכם אמור להביא לו אותו.

בדרך כלל, מתכווני צד השירות כתובים API לצד השירות כדי שיחזר לכם מידע, והאחריות של מתכווני ריאקט היא לשלוח בקשות לשרת. כך, למשל, אתם משתמשים בראוטינג דף התחברות עם שם משתמש וסיסמה. המשתמש מקליד שם משתמש וסיסמה ולוחץ על כפתור "שלח". האחריות שלכם היא לשלוח, באמצעות XAJA, את שם המשתמש והסיסמה ל-API של השירות. צד השירות אמר לחזר לכם תשובה של 200 וובייקט המכיל את פרטי המשתמש, בהנחה שהכל מצליח, ולשתול עוגיה הצד המשתמש. האחריות שלכם היא לחת את המידע הזה ולהכניס אותו לكونטיקט (למשל) או לרידקס, שנלמד בהמשך, כדי שכל המידע באתר יוצג למשתמש המחבר כמו שצריך.

כמו בראוטינג, ריאקט אינה קובעת עבור המשתמש איך לשלוח את הבקשות ואייר לנוהל אותן. אפשר לבצע את הבקשות בעזרת fetch הבלתי שיש בג'אוועסקריפט בסביבת הדף או בעזרת ספרייה. הספרייה הפופולרית לניהול בקשות XAJA היא Axios. אנו נתרגל באמצעות fetch. יש כמה דרכים לנוהל את הבקשות – אפשר לכתוב אותן בתוך הקומוננטה ממש. אנו נתאמן עם ה-API של בדיחות צ'אק נוריס, שבעצם מאפשר לנו לקבל בדיחות צ'אק נוריס בקישור הבא:

<https://api.chucknorris.io/jokes/random>

נסו בעצמכם! העתיקו והדביקו את הקישור בדף וצפו ב-JSON שmagiu.

אייר נממש את זה? כאמור – אפשר בקומוננטה:

```
import React, { useState, useEffect } from 'react'
```

```
function Welcome() {
```

```
const [joke, setJoke] = useState('Joke is loading...');

function getJoke() {
  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 קריית ה-AJAX, באמצעות `fetch`, אל פונקציה מסודרת שאנו מפעילים רק פעם אחת, באמצעות ההוק `useEffect` שמקבל כפרמטר ראשון את הפונקציה להפעלה וכפרמטר שני מערך ריק, שבעצם אומר שהוא יופעל רק כאשר הקומפוננטה מוחנחת. הפונקציה שעשוה גם מכינה את הבדיקה המתבקשת אל הסטייט.

הקוד הזה, שאמור להיות מובן מאוד אם אתם כבר מכירים AJAX וידעים איך עובד, הוא קוד ולידי, אבל הוא רעיון גרוע מאוד אם האפליקציה שלכם מעט יותר מורכבת, למשל אם יש דרישות נוספות כמו להכניס את תוצאות הבדיקה לקונטנסט, כדי שעוד קומפוננטות יכולו להציג אותה, או לשנות את ה-API. זו הסיבה שמקובל מאד להכניס קריאות לשרת לפונקציות ג'אווהסקריפט טהורות שנקרוjas Services (Services). יוצרים בפרויקט תיקייה 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() {
  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 React, { useState, useEffect } from 'react'
import getChukJoke from './services/chuckJokes';

function Welcome() {
  const [joke, setJoke] = useState('Joke is loading...');

  useEffect(getJoke, []);

  function getJoke() {
    getChukJoke().then((joke) => {setJoke(joke);});
  }

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

export default Welcome;
```

יש שירותים שנבנים כקלאסים ויש כאלה שנבנים כפונקציות. כל דרך היא טובה – אבל כדאי לשים לבשלא משנה איזו דרך נבחרה – מדובר בג'אווהסקריפט טהור. ריאקט לא מתערבת בענייני הסקיופים וכל אחד יכול למשוך אותם לפי הבנותו. סביר להניח שכאשר תעבדו מול מערכות אמיתיות, מי שיגדר לכם איך הסקיופים מתנהלים מול השירות יהיה מתכווני הבק אנד, והם אלו שיגדרו לכם איך להתחבר.

תרגילים:

נדרשתם להציג בדיחת אבא רנדומלית באתר. מתכונת צד השרת הגדייר לכם API באופן הבא: יש לשלוח בבקשת GET אל הכתובת:
<https://icanhazdadjoke.com>

עם בקשת ה-GET דואגים לשלוח את ה-*header* `:header` `Accept: application/json` אל הכתובת:
 כדי לקבל את בדיחת האבא.

רמז: בקשת `fetch` עושים באופן הבא:

```
return fetch(`https://icanhazdadjoke.com` , {
  headers: {
    'Accept': 'application/json'
  },
})
```

פתרונות:

```
export default function getDadJoke() {
  return fetch(`https://icanhazdadjoke.com` , {
    headers: {
      'Accept': 'application/json'
    },
  })
  .then((response) => {
    return response.json();
  })
  .then((jsonObject) => {
    return jsonObject.joke;
  });
}
```

משתמשים ב-service זהה כך:

```
import React, { useState, useEffect } from 'react'
import getDadJoke from './services/dadJokes';

function DadJoke() {
  const [joke, setJoke] = useState('Joke is loading...');

  useEffect(getJoke, []);

  function getJoke() {
    getDadJoke().then((joke) => {setJoke(joke);});
  }

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

export default DadJoke;
```

בגדיול, זה לא ריאקט אלא ג'אויסקורייפט. הSERVICE הוא פונקציה שמייצרת בבקשת AJAX. הבקשה נשלחת עם header ומחזירה אובייקט JSON. אנו לוקחים את האובייקט ומוחזרים את ה-joke מתוכו.

הפרויקט פשוט קורא לפונקציה שמחזירה פרומיס וכאשר הוא מתמלא, אנו מכניסים את מה שהוא נותן לנו לסתיט של joke.

בשלב זה אנו יכולים לטפל, כמובן, בשגיאות או בפלט לא צפוי. למשל, משהו בסגנון הזה:

```
import React, { useState, useEffect } from 'react'
import getDadJoke from './services/dadJokes';

function DadJoke() {
  const [joke, setJoke] = useState('Joke is loading...');

  useEffect(getJoke, []);

  function getJoke() {
    getDadJoke()
      .then(
        (joke) => { setJoke(joke); },
        () => { setJoke('Sorry! Error!'); }
      );
  }

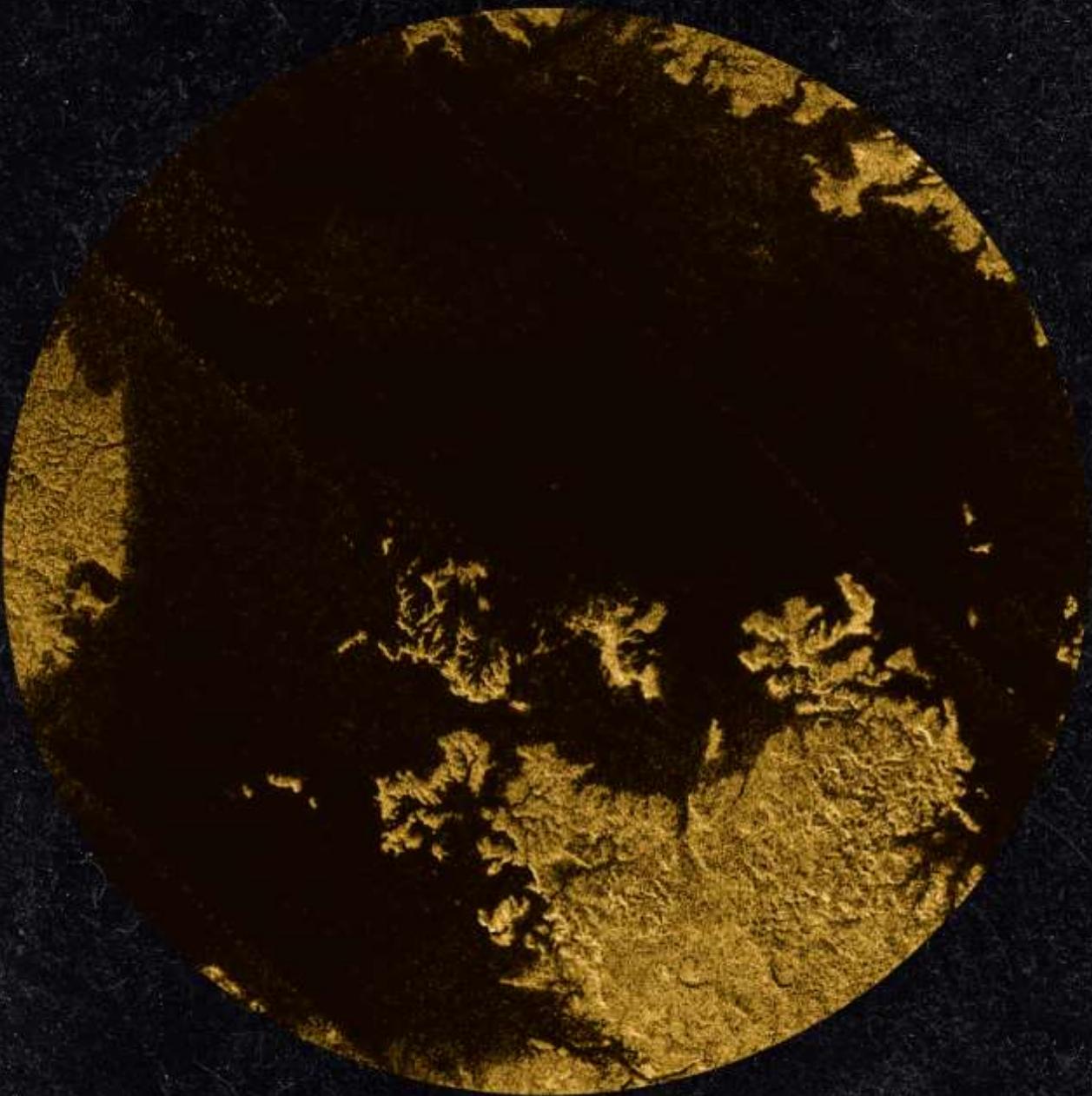
  return (
    <div>
      {joke}
    </div>
  );
}

export default DadJoke;
```

או כמובן ניהול מסודר יותר. חשוב מאד, בקריאה AJAX, לחשב על מה שקרה אם השירות לא מחזיר את התשובה המczופה.

פרק 22

מבוא לבדיקות נם JEST



מבוא לבדיקות עם jest

בדיקות אוטומטיות הן חלק חשוב מפיתוח מקצועי. בפיתוח יש לנו כמה סוגים של בדיקות יחידה (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) שבבודקות את כל האפליקציה מckaה לckaה.

בדיקות חיוניות לפיתוח נכון, וברוב החברות הטובות נהוג לכתוב בדיקות יחידה. הבדיקות הללו קרייטיות כיון שהם מתכונת אחר יעשה شيئا' שונה בקומפוננטה שלכם, הבדיקה תישבר והוא יビין שיש בעיה עם תסريع מסוים שבניתם עליו ובדקתם אותו. זו הסיבה שבדיקות הן דבר חשוב ולא מיועד רק לאנשי QA. בעולם הפיתוח מקובל שפתחים כתבים בבדיקות יחידה.

בדיקות אוטומטיות הן סקריפטים, הכתובים בג'אוויסקייפ, ויש לכל בדיקה קרייטריונים להצלחה. למשל: "אם לוחצים על כפתור, האפליקציה קוראת לשירות". הבדיקות האלו רצויות באופן נפרד באמצעות פקודה מסוימת ומריצים אותה לפני כל הכנסת קוד חדש. בגלל זה בבדיקות אוטומטיות הן חלק ממה שנקרא Continuous Integration, שהוא תהליך שקיים חדש עבור כשזהו משולב בקוד שנכתב קודם לכן. אם מישחו כתוב תוספה לקוד שלנו (למשל הוסיף props נוספים) ויש לנו בבדיקות טובות שרצות על הקוד החדש וכל הבדיקות רצויות אף אחת מהן לא נכשלת – אנו יודעים שכנהראה נכתב קוד איקוטי (או לפחות צזה שעבר את הבדיקות) ושנוכל להעלות אותו לאויר.

ספריית התוכנה ש谋ריצה את הבדיקות בריאקט נקראת `jest`. היא כבר מגיעה עם Create React App באופן מובנה. משתמשים ב-`jest` במקומות נוספים למעט ריאקט והיא ספרייה פופולרית מאוד לבדיקות. בבדיקות מקובל שקובץ הבדיקות נושא את השם של הקובץ המקורי עם התוספה `spec` או `test`. למשל, קובץ הבדיקות של `Welcome` יהיה `Welcome.spec.js`, אבל יש קונבנציות נוספות.

הבה נכתוב בדיקה אוטומטית ראשונה עבור `jsx.Welcome`, שתיראה כך:

```
import React from 'react'

function Welcome() {
  return (
    <p>Hello world!</p>
  );
}

export default Welcome;
```

אפשר לראות שזו קומפוננטה פשוטה מאוד. יש לה רק תפקיד אחד, להציג `Hello world!`. איך נבדוק אותה?

נכתוב 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.js`.

בתוכו אנו עושים `import` לריאקט ולרכיב שנקרא `react-dom`, שיסייע לנו לрендר את הקומפוננטה. אנו נבדוק את תכונת `innerHTML` של הקומפוננטה כדי לראות שהיא הדpisa את הפלט כמו שצריך. כך עושים זאת זה:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Welcome from './Welcome';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<Welcome />, div);
  expect(div.innerHTML).toEqual('<p>Hello world!</p>');
});
```

בואו נעבור על חלק הבדיקה.

חלק ראשון – import

אנו מייבאים את ריאקט (היא נדרשת על מנת שנוכל לрендר קומפוננטות) ואת react-dom, ומובן את הקומפוננטה שלנו. אלו החלקים הכי חשובים לבדיקה.

חלק שני – כתיבת מסגרת הבדיקה

זו הוא חלק חשוב מאוד בבדיקות והוא בא כאובייקט גלובלי מ-jest. הוא מקבל שני ארגומנטים: האחד הוא שם הבדיקה והשני הוא פונקציית חץ שבתוכה הבדיקה.

חלק שלישי – הרצת הקומפוננטה

הצעד הראשון הוא ליצור HTML שבתוכו הקומפוננטה שתתרנדר. מדובר בג'אוויסקריפט טהורה שבה יוצרים div סטמי ומקבלים רפרנס אליו:

```
const div = document.createElement('div');
```

הצעד השני הוא להשתמש ב-react-dom על מנת לрендר את הקומפוננטה. הפונקציה ReactDOM.render מקבלת שני ארגומנטים: האחד הוא הקומפוננטה, שלה עשוינו import, והשני הוא המיקום שבו היא מתրנדרת, במקרה זהה ה-div שיצרנו.

חלק רביעי – הבדיקה

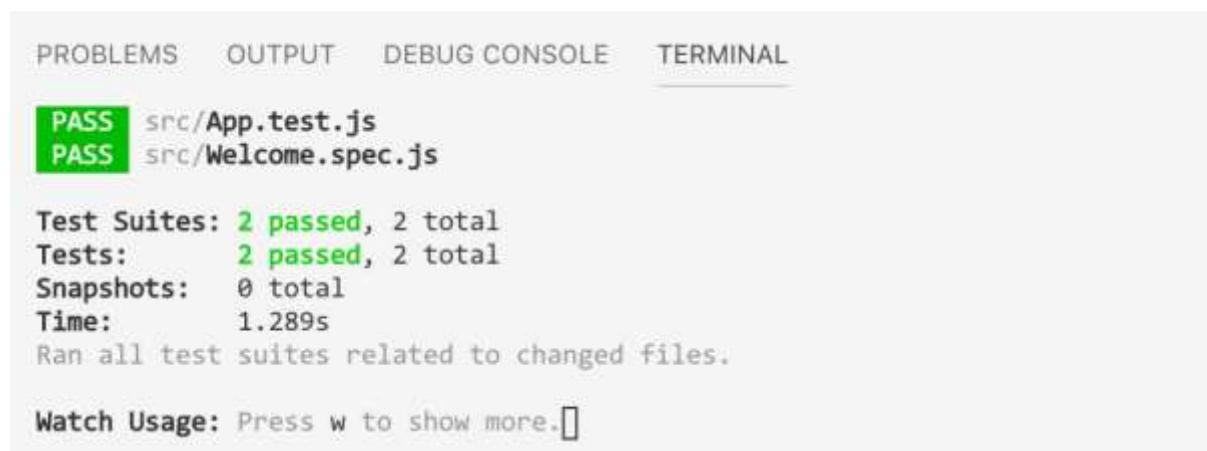
אחרי שהקומפוננטה רונדרה, אנו יכולים לבדוק את ה-div שבתוכו היא נמצאת. אנו עושים את הבדיקה באמצעות פונקציית expect. מדובר בפונקציה שגמ היא באה עם jest והיא מקבלת ביטוי שאליו יש לשרשר את ההנחה. במקרה זה הביטוי הוא `div.innerHTML` וההנחה היא `toEqual` – כלומר "הביטוי שווה ל...". כל בדיקה היא בעצם ביטוי והנחה. ההנחה היא מתודה שאליה אנו צריכים להעביר את הטקסט שאנו משווים אליו.

האמת היא שבמקרה זה ה叙述 הרבה יותר מסורבל מהקוד:

```
expect(div.innerHTML).toEqual(`<p>Hello world!</p>`);
```

הקוד הזה מדבר בעד עצמו והוא מובן מאוד. אחרי שכתבנו את הבדיקה הגע הזמן לראות שהיא עברה בהצלחה. את הבדיקות אנו מרכיבים בטרמינל שכבר למדנו לעבד איתו בפרקים קודמים. אם אנו עובדים עם Visual Studio Code, נפתח חלונית של טרמינל באמצעות בחירה בלשונית Terminal בחלק העליון של התוכנה ואז ב-New Terminal.

לחופין, אתם יכולים לפתוח cmd ולנוט אל תקיות הפרויקט. לא משנה באיזו דרך בחרתם, על מנת להריץ את כל הבדיקות יש להקליד: test npm. מיד יירץ התהילר ואם הבדיקה מעבור בהצלחה, תראו אישור.



The screenshot shows the Visual Studio Code interface with the 'TERMINAL' tab selected. The terminal window displays the following output:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PASS src/App.test.js
PASS src/Welcome.spec.js

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.289s
Ran all test suites related to changed files.

Watch Usage: Press w to show more. []

```

אם הבדיקה לא תעבור בהצלחה, תוכלו לראות את הסיבה שבגלה היא נכשלה:

```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PASS  src/App.test.js
FAIL  src/Welcome.spec.js
  • renders without crashing

    expect(received).toEqual(expected) // deep equality

      Expected: "<p>Hello World!</p>"
      Received: "<p>Hello World!</p>"

    6 |   const div = document.createElement('div');
    7 |   ReactDOM.render(<Welcome />, div);
  > 8 |   expect(div.innerHTML).toEqual('<p>Hello World!</p>');
      ^
    9 | });

      at Object.toEqual (src/Welcome.spec.js:8:25)

Test Suites: 1 failed, 1 passed, 2 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:  0 total
Time:        3.021s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.□

```

כאן לדוגמה בכתיבת הבדיקה ובמקום לבדוק שהקומפוננטה מדפסה Hello world! ב-A קטנה, הנחתת שהיא מדפסה Hello World! ב-W גדולה. רואים את השוני.

בדיקות לkomponentה עם props

אם לkomponentה יש props, علينا להעביר אותם בבדיקה. זה קל יותר ממה שזה נשמע. הנה נבחן את komponentה זו:

```
import React from 'react';

function Welcome(props) {

  return (
    <p>Hello, {props.name} {props.surName}!</p>
  );
}

export default Welcome;
```

כאן נעשה שימוש ב-`sprops`. כיצד אבדוק את komponentה? ראשית, אוצר את קובץ הבדיקות. כמעט דבר לא משתנה מלבד הרנדור של komponentה.מן הסתם אני ארצה לрендר אותה עם ה-`sprops` המתאיםים. איך עושים את זה? כך:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Welcome from './Welcome';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<Welcome name="Moshe" surName="Cohen" />, div);
  expect(div.innerHTML).toEqual('<p>Hello, Moshe Cohen!</p>');
});
```

השני העיקרי הוא השורה זו:

```
ReactDOM.render(<Welcome name="Moshe" surName="Cohen" />, div);
```

אם אני בוחן `sprops`, אני נדרש להעביר אותם ולבוחן את התוצאה. ברוב הפעמים נשתמש בספריית עזר לשם הבדיקה, כמו ספריית העזר Enzyme שמשיעת מאוד בבדיקה אירועים וברנדור של komponentות והkomponentות הבנות (היא לא נלמדת במסגרת הספר).

פונקציית בדיקה – assert

בבדיקה שעשינו לשם הדוגמה, השתמשנו ב-`toEqual`. כלומר, משווים ממש את הפלט. הקוד שבי השתמשנו הוא:

```
expect(div.innerHTML).toEqual('<p>Hello, Moshe Cohen!</p>');
```

הוא מורכב מכמה חלקים. החלק הראשון הוא פונקציית `expect`, שmagie עם `jest` ומקבלת ערך כלשהו, טקסטואלי או לא. החלק השני הוא `toEqual`, שמבצעת את ההשוואה. ההשוואה זו נקראת פונקציית `assert`, ומספרית `jest` מציעה לנו לא מעט פונקציות בדיקה כאלה.

not

כשאנו מוסיפים את הפונקציה `not` לפני פונקציית ההשוואה, אנו בעצם הופכים אותה. הבדיקה הזאת, למשל, בודקת שהטקסט הוא לא `Error`:

```
expect(div.innerHTML).not.toEqual('Error');
```

toBeTruthy

פונקציה שבודקת שיש לנו ערך שהוא לא `undefined`. זה חשוב כאשר לא אכפת לנו איזה ערך אנו מקבלים, העיקר שייהי מדובר בערך אמיתי. בדרך כלל זה נעשה בבדיקות ראשוניות, כאשר אנו רק רוצים לוודא רנדור של קומפוננטה או ערך מסוים. למשל, אם אני רוצה לדעת שיש גובה לקומפוננטה, ולא אכפת לי איזה גובה.

```
expect(div.clientHeight).toBeTruthy();
```

toBeFalsy

פונקציה שבודקת שלא קיבלנו ערך. היא שימושית אם רוצים לדעת שבמקרה הזה הבדיקה מחזירה לנו משהו שלא קיים. למשל, אם אני רוצה לוודא שאין `id` לקומפוננטה – כיוון שכמה קומפוננטות עם אותו `id` יכולות לעורר בעיות ב-DOM – אני אבצע את הבדיקה באופן הזה:

```
expect(div.id).toBeFalsy();
```

toContain

פונקציה שבודקת שהפרמטר שאנו מעבירים לה נמצא בטקסט ההשוואה. זה שימושי כאשר אנו לא רוצים לבצע בדיקה מלאה. אם נסתכל על הדוגמה שלנו, מדובר למשל בבדיקה שהשם `Moshe` מופיע:

```
expect(div.innerHTML).toContain('Moshe');
```

תרגילים:

נתונה הkomponenta Greeting. כתבו לה בדיקה שבודקת שהיא מדפיסה את הפלט name כשמותibus ליה את ה-name המתאים.

```
import React from 'react';

function Greeting(props) {

  return (
    <p>Good Morning {props.name}!</p>
  );
}

export default Greeting;
```

פתרונות:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Greeting from './Greeting';

it('Output the correct name', () => {
  const div = document.createElement('div');
  ReactDOM.render(<Greeting name="Moshe" />, div);
  expect(div.innerHTML).toEqual('<p>Good Morning Moshe!</p>');
});
```

בדוק כמו בדוגמה שהובאה בפרק, ייבנו את הקומוננטה ובאמצעות ReactDOM.render ייצרנו אותה בתוך ה-div. הבדיקה האםית היא השוואת הפלט של הקומוננטה באמצעות div.innerHTML. את הרצה אנו מבצעים באמצעות run test npm.

הפלט יראה כך:

```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL

PASS  src/components/Greeting.spec.js
  ✓ renders without crashing (21ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.504s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.

```

תרגום:

שנו את הבדיקה ובדקו רק שהטקסט Good Morning נמצא שם, ללא הפרמטרים.

פתרון:

```

import React from 'react';
import ReactDOM from 'react-dom';
import Greeting from './Greeting';

it('Contains Good Morning', () => {
  const div = document.createElement('div');
  ReactDOM.render(<Greeting name="Moshe" />, div);
  expect(div.innerHTML).toContain('Good Morning');
});

```

אנו משתמשים במקרה זה בבדיקה של `toContain`, כיוון שהיא שמעניין אותנו הוא המילה הזו ולא כל הבדיקה. בדרך כלל נשתמש ב-`toContain`, כיוון שאם נשנה את המבנה הפנימי של הקומפוננטה לא נשבור את הבדיקה.

תרגום:

נתונה הקומפוננטה `xsj`.
Random שמדפסה מספר רנדומלי:

```

import React from 'react';

function Random() {

  return (
    <React.Fragment>{Math.random()}</React.Fragment>
  );
}

export default Random;

```

כתבו בדיקה שבודקת שהקומפוננטה מוציאה פלט מסוים.

פתרונות:

```

import React from 'react';
import ReactDOM from 'react-dom';
import Random from './Random';

it('Contains Good Morning', () => {
  const div = document.createElement('div');
  ReactDOM.render(<Random />, div);
  expect(div.innerHTML).toBeTruthy();
});

```

כתיבה הבדיקה הזאת היא פשוטה, כיון שמדובר בקומפוננטה מאוד לא מסובכת. אבל כשאנו רוצים לבדוק את הפלט שלו אנו בבעיה, כי הפלט משתנה בכל פעם. אנו לא יכולים להשתמש בפונקציית `toEqual` או `toContain` כי הקומפוננטה מייצרת בכל פעם מספר רנדומלי. מה הפתרון? להשתמש ב-`toBeTruthy` כדי לוודא שיש משהו ב-`innerHTML`.

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 יומם כל האפליקציה היא קובץ ג'אווהסקריפט, קובץ HTML וקובץ CSS שנטענים על ידי הדפדפן. בקובץ הג'אווהסקריפט זהה, הджוס והמחובר, יש הכל: ספריית ריאקט, babel, הקומפוננטות החיצונית והקוד שככבותם. הכל בקובץ אחד. הדפדפן טוען אותו והכל מתחילה להיבנות ממש. יש כאלה שבוחרים לחלק את הקובץ הדחוס לכמה חלקים – למשל תלויות חיצונית (ספריות צד שלישי) כמו למשל UI React Material Sheulia למדנו באחד הפרקים הקודמים) והאפליקציה עצמה שככבותנו. אבל בסופו של הבילד, התוצאה היא קבצים דחוסים.

הקובץ הזה אמר להגיא לדף איכשהו. כשאנו בסביבת הפיתוח, עם קומון Create React Applocalhost:3000, יוצרת לו כתובות (mbosso.js.Node), ופתחת את הדפדפן ומפנה לאתר הזה שטוען בתורו את קובץ הג'אווהסקריפט. אבל מה קורה באתר אמיתי?

באתר אמיתי אנו צריכים שרת אמיתי שיגיש את הקבצים לדף. כשאני אומר "יגיש" אני מתכוון לתהילך שבמסגרתו אנו מקלדים כתובות של אתר אינטרנט בדף. הדף ניגש לשרת האתר והוא מחזיר לנו קבצים שהדף מציג.

כך למשל כשאני גולש לגוגל, אני מקליד בדף google.com, השירות 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. קובץ הג'אווהסקריפט המכוחץ של האפליקציה.

איך אנו יוצרים את השירות הזה? יש הרבה דרכים. הדרך פשוטה ביותר היא לשכור shared hosting לאתר כלשהו ופשוט להציב עליו את קובץ index.html וקובץ CSS והג'אווהסקריפט (זה עובד אם עושים ראותינג מבוסס #), באמצעות העתקה פשוטה, בתיקייה הראשית של השירות בעזרת הממשק של ניהול השירות שספק השירות מעניק לכם. הקולדת כתובות שם המתמחם וכניסה אל האתר תגרום לטעינת קובץ HTML וכמובן כל הקבצים המוקשרים אליו.

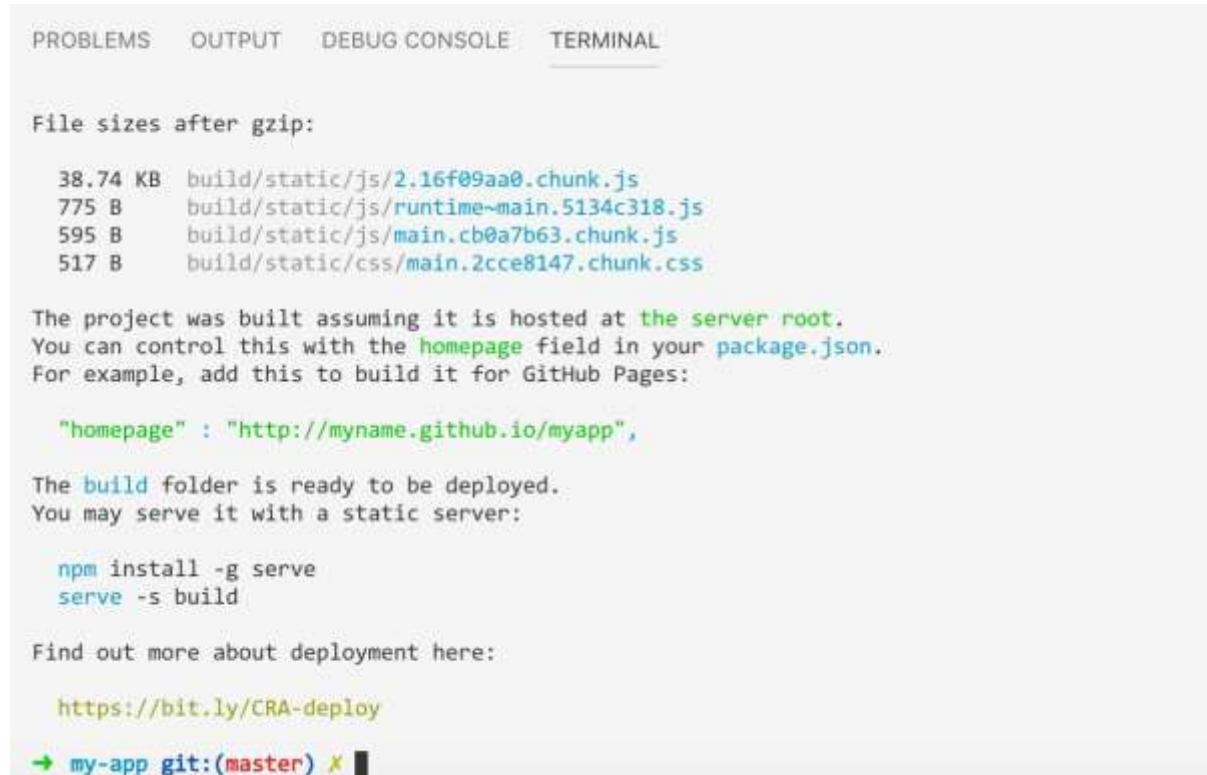
יש דרכים רבות אחרות – לא חסרות שפותצד שרת שיש בהן שירותים. הפופולרית ביותר היא js.Node, אך יש גם את ג'אווה, #C ופיתוח SMAF שמאפשרות ייצור שרתים. אם אתם או מוכנתםצד השירות בחברה שלכם, יכולים יוצרים שרת זהה – בסופו של דבר תצטרכו לגורם לו להחזיר לךו את דף HTML והקבצים. הלקוח יכנס, הקבצים ייטענו ומפה תהפוך האפליקציה לפועלה בדיקון כמו ב-AppCreate React App.

AIR מייצרים את קובץ ה-HTML, קובץ ה-CSS וcmbn קובץ ה'אואוסטקריפט? באמצעות פקודת npm מיוחדת שיש להקייש בטרמינל – משך הפקודה הטקסטואלי. אם אתם משתמשים ב-Visual Studio Code, לחצו על Terminal בתפריט העליון ואז על New Terminal. יפתח לכם חלון בתחום המסר עם שורת פקודה טקסטואלית.

אם אין לכם Visual Studio Code, היכנסו אל ה-cmd בחלונות ונותנו אל התיקייה של האפליקציה. מיד כשאתם יכולים להקליד בטרמינל, הקלידו:

```
npm run build
```

ירוץ תהליך שמאחד ומכooץ את קובצי ה-**HTML**, **CSS** וה-**JS**(index.html) וтвор קובץ קבצים סטטיים נוספים שימושיים לשרת. כל הקבצים האלו ייכנסו אל **תיקיית build** – זו התיקייה שיש להעלות אל השרת כדי להשתמש באפליקציה שלכם, בין שמדובר בשרת של חברה גדולה ובין שמדובר ב-**:shared hosting**



The screenshot shows a terminal window with tabs: PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is active.

```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

File sizes after gzip:

38.74 KB  build/static/js/2.16f09aa0.chunk.js
775 B     build/static/js/runtime~main.5134c318.js
595 B     build/static/js/main.cb0a7b63.chunk.js
517 B     build/static/css/main.2cce8147.chunk.css

The project was built assuming it is hosted at the server root.
You can control this with the homepage field in your package.json.
For example, add this to build it for GitHub Pages:

  "homepage" : "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

  https://bit.ly/CRA-deploy

→ my-app git:(master) ✘
  
```

התהליך הזה נקרא **билד** והוא נפוץ באפליקציות צד לקוח כמו ריאקט, אングולר ודומותיהן. התהליך זהה מתרחש אוטומטית כאשר מתכנת מוסיף קוד לפרויקט והתהליך משולב בדרך כלל עם הבדיקות האוטומטיות.

אפשר לבדוק את הבילד באמצעות מודול `serve` שנקרא `serve`. מתקנים אותו גלובלית באמצעות הקלהת הפקודה זו בטרמינל:

`npm install -g serve`

ולאחר מכן:

`serve -s build`

וכניסה אל:

localhost:5000

ההבדל בין השיטה זו להקלהת start וקח הוא שכאן אנו מרכיבים את הבילד ממש, כלומר את הקבצים המוגמרים שאנו אמורים להעלות לשרת. מודול serve יוצר לנו שרת ומעלה את התיקייה build כדי שנוכל לבדוק את האפליקציה שלנו, מתיקית ה-build, מקומית.

שימוש לב: אם כאשר אתם מקלידים serve אתם מקבלים הודעה שגיאה בנוסח:

```
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
```

תרגילים:

קחו כמה קומפוננטות שיצרנו בתרגילים הקודמים ושלבו אותן באפליקציית הריאקט שיש ב-sj.App ב-Create React App. הריצו את הבילד ובדקו אותו.

פתרונות:

יש להרץ build run npm ולחכות לתוכאה. אחרי שהתהליך מושלם, תיווצר אצלכם תיקיית build עם קוד HTML, קובצי תמונה, CSS והכǐ חשוב: ג'אווהסקריפט.

על מנת לבדוק את הבילד, אפשר להשתמש ב-node serve מקומי כפי שלמדנו. תיפתח לכם בדף אפליקציה מלאה שנמצאת ב-localhost.localhost. אפשר להעלות אותה לכל שרת באמצעות העתקת תיקיית build והעברתה לשרת מבוסס Node.js, Apache או כל שרת אחר.

פרק 24

ליזה



רידק

חלק שימושותי בהבנה של ריאקט הוא נושא ניהול הסטייט בקומפוננטה - מתי מעדכנים את הסטייט? מתי מוצאים אותו החוצה? מתי מעבירים אותו לקומפוננטת בת? בחלק מהקומפוננטות יש סטייט פנימי – אבל הרבה פעמים יש לנו צורך בסטייט משותף. למשל בדף שיש בו המון קומפוננטות שצריכות לקבל אובייקט עם פרטי המשמש, קומפוננטה אחת תציג ברכבה, "שלום רן", קומפוננטה אחרת תציג קישור למודול (פוף-אפ) פרטי משתמש שבו יש קומפוננטה שאמורה להכיל את אובייקט המשמש. קומפוננטה נוספת, בתחתית הדף, צריכה לקבל גישה לאובייקט כדי לדעת אם להציג פרטיים משפטיים מסוימים ללקוח, וכך הלאה. יש הרבה מאוד קומפוננטות שצריכות גישה לפרטים האלו.

כדי שהשמה תהיה כפולה – עליה השאלה מה קורה אם הסטייט מתעדכן. למשל אם משתמש שינה את שם המשתמש שלו? או, למשל, אם יש מונה הודעות – מה קורה אם הקורא קרא הודעה אחת ויש להויריד את מונה ההודעות?

זו הסיבה שאנו זקוקים לסטייט משותף של האפליקציה, סטייט שאפשר לעדכן מכל קומפוננטה וסטייט שאפשר לקרוא מכל אפליקציה. מצד שני, חשוב לשמור על הסדר, כדי שלא כל שינוי בסטייט המשותף ירנדר את כל הקומפוננטות, גם כאלה שלא משתמשות בחלק מהסטייט השנתה; וכן שמי שירצה לשנות את הסטייט יוכל לעשות את זה, אבל רק את החלק שלו.

בפרק על קונטיקסט הסבכנו על סטייט משותף והראנו איך עושים את זה עם קונטיקסט ועם הוקים. ללא ספק זו הדרך שריאקט מתקדמת אליה, אבל נכון לגרסה הנוכחיית, קונטיקסט הוא לא תחילה לסטייט משותף מכמה בחינות, במיוחד ביצועים: קונטיקסט לא נועד לשינויים תכופים אלא להעברת מידע סטייט (או מידע ששואף להיות סטייט) כמו טמפליט, אובייקט משתמש וכו'. לעומת זאת, ספריית RIDK (Redux), ספרייה חיצונית שלילית נלמד בפרק זה, מוכונת לדברים שימושיים כל הזמן. נוסף על כן, לרידק יש שני פיצרים שימושיים מאוד: מסע בזמן, שמאפשר לנו, באמצעות כל הפיתוח של RIDK, לזרז לאורך חיים האפליקציה ולראות מה השתנה ולשנות אותו בפועל, ו-middlewares – שכבת תוכנה נוספת בין הנtauונים לאפליקציה.

כיוון שניהול סטייט עומד פעמים רבות בפני עצמו ואין קשר בהכרח לריאקט, יש כמה ספריות לניהול סטייט משותף – הראשית והחשובה שבahan היא RIDK. אפשר להשתמש ברידק גם בפרימורוקים אחרים והיא חיצונית לריאקט, אבל בשנים האחרונות היא הפכה למזהה מאוד עם ריאקט.

RIDK נחשבת למורכבות משתי סיבות עיקריות: הראשונה היא הקונספט – הבנת הקונספט של מה RIDK עשה ואייר היא מנהלת סטייט משותף. השנייה היא הקוד – תוספת הקוד של RIDK היא שימושית ומסורבלת. כדי שקומפוננטה תוכל להשתמש ברידק יש לעתוף אותה בקוד RIDK זהה מורכב. מעבר לסרבול הקוד, RIDK גוטה לתဏוף ולהכיל גם סטייטים שלא אמרוים להיות גלובליים.

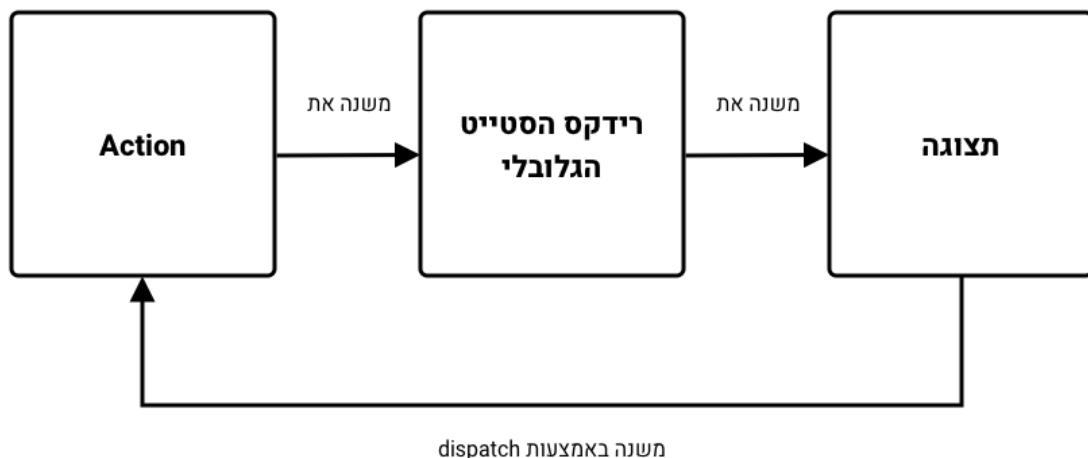
לכן אני ממליץ להשתמש בקונטיקסט באפליקציות קטנות יותר וברידק באפליקציות גדולות יותר, במקום שתוספת הקוד והמורכבות שווה יותר.

הפילוסופיה מאחורי RIDKOS

RIDKOS דוגלת בשלושה עקרונות עיקריים:

1. מקור אחד של אמת (Single Source of Truth) – יש סטיטי אחד ויחיד והוא הנקודה שבה יש את המידע האמתי והאמין ביותר.
2. הסטיטי הוא קרייה בלבד – אי-אפשר לשנות את הסטיטי באופן ישיר. אפשר רק לקרוא ממנו.
3. שינויים נעשים באמצעות פונקציות טהורות – אם אנו רוצים לשנות את הסטיטי, אנו עושים זאת באמצעות פונקציות שאין להן השפעות צדדיות ובהינתן אותו קלט, עושות בדיקת את אותה הדבר.

אם ננסה להמחיש את זה באמצעות תרשים, נראה שיש לנו סוג של מעגל עם כיוון אחד של זרימת מידע. אנחנו מתחילה בפעולה, שモפעלת על ידי קומפוננטה כלשהי באמצעות dispatch. הפעולה הזאת משנה את הסטיטי הגלובלי. אף אחר, חוץ מאותו חסияן, לא יכול לשנות את הסטיטי. הסטיטי בתורו משנה את התצוגה ואז המש坦ש יכול שוב לעשות פעולה שתמשיך את המעגל.



כרגע זה נשמע מאוד אבסטרקטי, אבל כשנלמד איך מיישמים RIDKOS בפועל, נראה איך העקרונות האלה מתממשים.

קומפוננטות ניהול מול מנהלות

כשאנו מסתכלים על קומפוננטות בריאקט, הן מחולקות לשני סוגים:

קומפוננטות תצוגה או קומפוננטות מנהלות – קומפוננטות שלא מודעות לרידקס ולא משתמשות בו. קומפוננטות אחרות מנהלות אותן באמצעות `props` ומקבלות מהן מידע באמצעות אירועים. הן מגדרות איך מידע נראה.

קומפוננטות מנהלות – קומפוננטות שמודעות לרידקס ועטופות על ידו. הן מנהלות קומפוננטות אחרות ומעבירות אליהן את המידע שmagick מרידקס. הן מקבלות מידע מרידקס על ידי רישום לאירועי RIDKES ומשנות מידע על ידי `dispatch` (הפעלה) של פעולות (`actions`) RIDKSIOT. הן מגדרות איך מידע מתנהל.

קומפוננטות ניהול**קומפוננטות ניהול**

מתמקדות באיך דברים עובדים

מתמקדות באיך דברים נראהים

מודעות לרידקע

לא מודעות לרידקע

רשומות לסטיט הרידקע'

קוראות למידע מה-props

משתמשות ב-`dispatch` על מנת להעביר פלט
הרידקעמשתמשות בקולבקים כדי להעביר פלט
הרידקע

נקודות על ידי רכיב רידקע' עוטף

נקודות ביד

כשאנו מתכננים את מבנה הקומפוננטות שלנו, אנו מבצעים את הפרדה בין הקומפוננטות המנהלות והקומפוננטות המנהלות, אלו שכנ עטופות על ידי רידקע'.

בדוגמה שלנו נדגים באמצעות אפליקציה של רשימת קניות. הינה התוכן הראשוני:

תפקיד	שם קובץ	מה היא אמורה לעשות
אפליקציית ריאקט	App.js	מכילה את קומפוננטת הקונטינר.
קונטינר	Container.jsx	מכילה קומפוננטה של <code>ListInput</code> שבה אפשר להכניס מידע לרשיימה.
קונטינר	Container.jsx	מכילה קומפוננטה של <code>ListView</code> שבה הרשימה מוצגת.
קומפוננטה המתקבלת כתלט	ListInput.jsx	אלמנט שוקה פשוט שבו משתמש מקליד את פרטי הפריט להכנסה לרשיימה.
קומפוננטה המציגת פלט	ListView.jsx	קומפוננטה המכניתה מידע לרידיקס.
אחרי שמייפינו את הקומפוננטות השונות我们知道 ונו יודעים אילו קומפוננטות יש לנו, זה הזמן לבנות את האפליקציה ללא רידיקס.		הציגת הרשיימה. קומפוננטה המתקבלת כתלט מידע מרידיקס.

ראשית, האפליקציה הריאקטית שנמצאת ב-`App.jsx`:

```
import React from 'react';
import './App.css';
import Container from './Container';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Container />
      </header>
    </div>
  );
}

export default App;
```

מדובר כמובן בשורש הקומפוננטות. כל תפקידה של האפליקציה הוא להביא קומפוננטת `Container` ולחדרנו עליה בעבר. קומפוננטת `App.jsx`:

```
import React from 'react';
import ListInput from './ListInput';
import ListView from './ListView';

function Container() {

  return (
    <div>
      <ListView />
      <ListInput />
    </div>
  );
}

export default Container;
```

אם לא היה לנו את רידקסט, הקומפוננטה הזו הייתה מורכבת מעט יותר, כיוון שהיא הייתה צריכה לנהל סטייט שה-`ListInput` היה מעדכן וה-`ListView` היה מקבל ומציג, או לעתוף את הקומפוננטות הללו בקונטקסט פרוביידר ולגרום להן לעשות את זה. כיוון שיש לנו רידקסט, אנו נותרים עם קומפוננטה פשוטה.

קומפוננטת `ListInput.jsx`

```

import React, { useState } from 'react';
import { TextField, Button } from '@material-ui/core';

function ListInput() {
  const [item, setItem] = useState('');

  function changeHandler(e) {
    setItem(e.target.value);
  }

  function clickHandler(e) {
    // Insert item to redux state
  }

  return (
    <div>
      <TextField onChange={changeHandler} />
      <Button onClick={clickHandler}>Insert item to list</Button>
    </div>
  );
}

export default ListInput;

```

זו הקומponeנטה שבה יש שדה טקסט וcptor (שניהם מгиים מספריית קומponeנטות חיצונית) וטיט פנימי משלה. כאשר לוחצים על cptor, מה שיש בסטייט item אמרור להגיא לסטיט הגלובלי של RIDKO.

המטרה שלנו: להכניס את סטייט item לרידקו.
קומponeנטת **ListView.js**

```
import React from 'react';
import { List, ListItemText } from '@material-ui/core';

function ListView(props) {
  // props.items should come from redux
  const listItems = props.items || [];
  const listItemsJSX = listItems.map(item =>
<ListItemText>{item}</ListItemText>);

  return (
    <List>
      {listItemsJSX}
    </List>
  );
}

export default ListView;
```

הkomponentה זו מציגה מערך של פריטים כרשימה. היא לוקחת את `props.items` (אם הוא לא קים, היא מציבה מערך ריק) ומmirה אותו לרשימה באמצעות הקומponentות החיצוניתות `List` ו-`.ListItemText`.

המטרה שלנו: שהkomponentה תקבל את `props.items` מרידקוט. עכשו אנו יכולים לgesת להטמעת RIDKO.

הטמעת RIDKOS

התקנת RIDKOS

RIDKOS היא ספרייה חיצונית ועליינו להתקין אותה באמצעות npm, שלמדנו לעילו בפרק על התקנת קומפוננטות חיצונית. ההתקנה היא פשוטה ומבצעת באמצעות הקילדת הפקוודה הבאה בטרמינל, במקום שלuproject שלנו:

```
npm install react-redux
npm install redux
```

תכנון הסטייט וזרימת המידע במערכת

הסטיט הגלובלי שלנו הוא בעצם אובייקט ג'אווהסקריפט פשוט שיש בו מערך שכול, תיאורטית, להכיל רשימה של דברים. למשל:

```
{
  items: []
}
```

הסטיט המשותף הוא אובייקט ג'אווהסקריפט רגיל שnitן לשינוי. במקרה זהה, יש לי תקונה בשם items שמכילה אובייקט ריק.

אחד העקרונות המרכזים של RIDKOS הוא שאי-אפשר לגעת בסטייט ישירות. אז איך אני משנה אותו? בעזרה action. למשל:

```
{ type: 'ADD_ITEM', payload: 'Item #1' }
```

הסוג הוא סוג שאינו מגדר, במקרה זהה הוספה של פריט לרשימה. ה-payload הוא תקונה שאני בוחר. זה יכול להיות payload או itemText או כל תקונה אחרת שאני בוחר. אני מחייב, אך מוקובל להשתמש ב-member שנקרה payload.

אחרי שהחליטתי על ה-action, אני צריך ליצור פונקציה שתיקח אותו ותכניס אותו אל הסטייט. חשוב מאוד להציג אובייקט חדש לסטיט כדי להראות לרידקוס שצריך לנדר הכל מחדש. לא מבצעים שינוי של הסטייט עצמו.

משתמשים בטריק פשוט כדי ליצור אובייקט חדש, במקרה זהה:

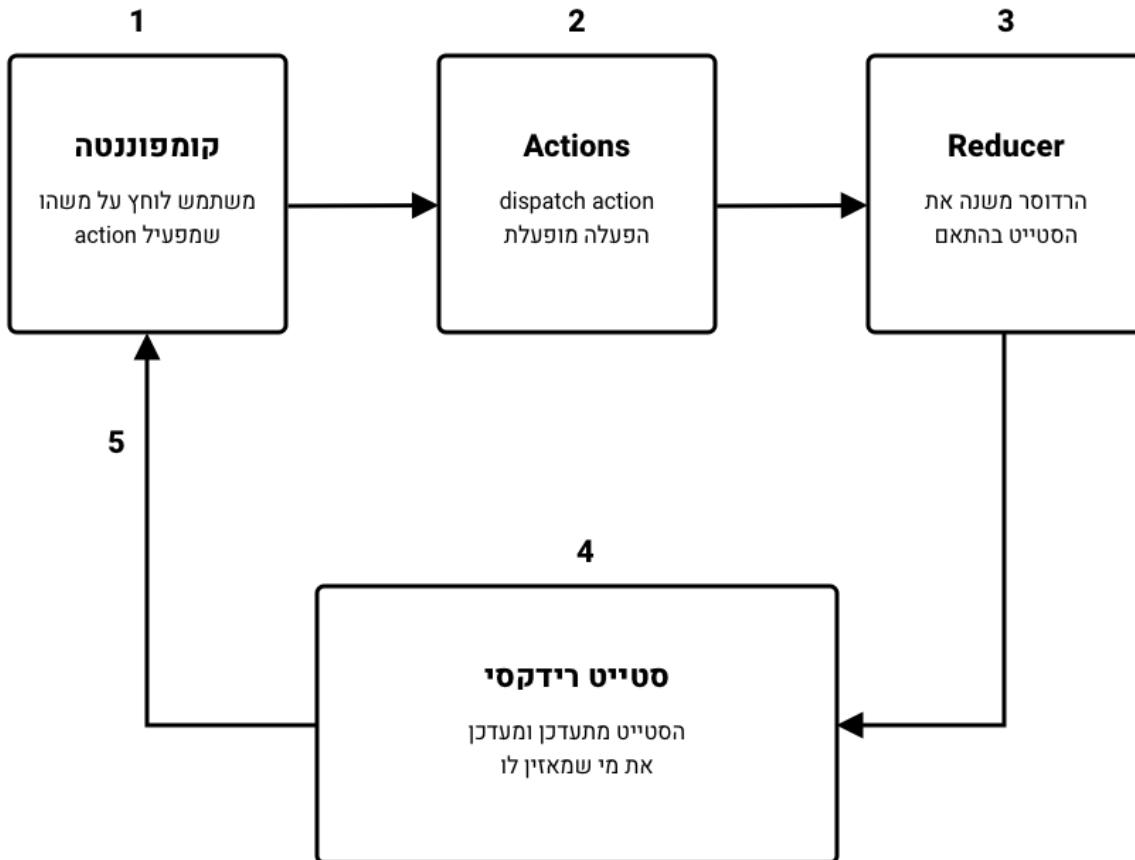
```
function manageList(state = { items: [] }, action) {
  switch (action.type) {
    case ADD_ITEM:
      const oldItems = state.items || [];
      return {
        ...state,
        items: oldItems.concat(action.payload),
      };
    default:
      return state;
  }
}
```

לפונקציה זו יש שם מפכיד: `reducer`. התפקיד שלה הוא לקבל action ולהחליט איך הוא משפיע על הסטייט. במקרה הזה, אם ה-`action` הוא מסוג `ADD_ITEM`, ה-`reducer` לוקח את ה-`action.payload` ומכוון אותו אל תוך הסטייט הגלובלי ומחזיר אותו. זה הכל. פונקציות ה-`reducer` הן פונקציות טהורות (pure functions), פונקציות ג'או-הסקרייפט שainן משנה את הקלט ומחזירות פלט קבוע על אותו קלט. כדאי לשים לב שחייב להיות `default` שמחזיר את הסטייט כפי שהוא.

از הינה ה-`use` הבסיסי: כשאנו רוצים להוסיף פריט לרשימה, אנו יורים את ה-`action`, ה-`reducer` תופס אותו ומעדכן את הסטייט הגלובלי. כמובן, כשאני רוצה להוסיף לרשימה אני קורא ל-`manageList` עם אובייקט מתאים. המטרה של `manageList` היא לטפל ברשימה. אם היא מקבלת אובייקט שנראה כך:

```
{ type: 'ADD_ITEM', payload: 'Item #1' }
```

היא תחזיר סטייט שהוא המערך הקודם עם `Item #1`+#. ואז הסטייט יתעדכן.



זה יכול להישמע מבלבל. אבל ברגע שambilים שניהול הסטייט הוא שונה מריאקט ומתבצע ב민ימום פעולה מריאקט זה מס'יע. החיבור נעשה באמצעות שתי פונקציות. אחת המחברת את הסטייט ל-props לקומפוננטות שמאזינן לסטט והשנייה הקוראת לפונקציות שמשנות את הסטייט.

mapStateToProps

בסיום של דבר, גם כשאנחנו משתמשים ברידקס, אנחנו כתבים קומפוננטות ריאקטיות.

1. קומפוננטה ריאקטית מקבלת data מבוחר באמצעות props.
2. אנחנו רוצים להעביר לקומפוננטה שלנו נתונים מהסטט.
3. אז בעצם אנחנו רוצים למפות חלקים מהסטט לתוך ה-props של הקומפוננטה שלנו.

mapDispatchToProps

הקומפוננטה שלנו מודיעה על שינויים באמצעות פונקציות שהיא מקבלת ב-props. אנחנו רוצים לגרום לשינוי בסטייט.

1. שינוי בסטייט נעשה באמצעות dispatch ל-actions.
2. אז אנחנו רוצים להעביר לקומפוננטה שלנו פונקציה שכשפעילים אותה היא יורה dispatch עם ה-action הנכון.

הימוש של רידקס במערכת שלנו

חשוב לציין, לפני שמתחלים, שמדובר בתהליך ארוך ולא פשוט. זו הסיבה שהמלצת היא לישם רידקס רק באפליקציות מורכבות. על אפליקציה קטנה עם כמה עשרות קומפוננטות בלבד – זו פשוט פעולה מורכבת מדי.

אנו נתחיל מלמעלה למטה. ראשית, ב-`index.js` של אפליקציית הריאקט עצמה:

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import { Provider } from 'react-redux';
import { createStore, compose } from 'redux';
import rootReducer from './redux/reducers';
const enhancers = compose(
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
  window.__REDUX_DEVTOOLS_EXTENSION__(),
);
const store = createStore(rootReducer, { items: [] }, enhancers);
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root'),
);
```

עשינו כאן כמה צעדים. הצעד הראשון הוא לבצע import לרכיבים הרידקסיים:

```
import { Provider } from 'react-redux'
import { createStore, compose } from 'redux'
import rootReducer from './redux/reducers'
```

הצעד השני הוא ליצור enhancer המאפשר לנו להשתמש בכל המפתחים של רידקס, שנלמד עליון בהמשך:

```
const enhancers = compose(
  window.__REDUX_DEVTOOLS_EXTENSION__ &&
  window.__REDUX_DEVTOOLS_EXTENSION__()
);
```

חשוב: את ה-`enhancer` מוסיפים בדרך כלל רק בסביבת פיתוח.

הצעד השלישי הוא ליצור את הסטיטייט הרידקס' הראשוני, שנקרא `store`. אפשר להתייחס אליו כאל בסיס נתונים גלובלי שמחזיק את כל הסטיטייט ולעתוף אליו את ה-`App` שלנו:

```
const store = createStore(rootReducer, { items: [] }, enhancers);

ReactDOM.render(<Provider store={store}><App /></Provider>,
  document.getElementById('root'));
```

הערה חשובה: אל תתבלבלו מהשם זהה. הפרוביידר כאן הוא של רידקס ואינו קשור לكونטקסט.

כעת אנו מוכנים לעבוד עם RIDKo.

השלב השני הוא ליצור את ה-actions, הפעולות שבוצעים להן dispatch. אנו ניצור תיוקית redux בפרויקט שלנו ותת-תיק"יה בשם actions.js. שם ניצור קובץ js.index שמכיל את ה-action שלנו, שכבר תכננו מוקדם יותר:

```
/*
 * action types
 */

export const ADD_ITEM = 'ADD_ITEM'

/*
 * action creators
 */

export function addItem(text) {
  return { type: ADD_ITEM, payload: text }
}
```

חשוב לדעת: ה-type action חייב להיות "יחודי".

הfonקציה זו היא פונקציה (בשם action creator) שקומפוננטות יכולות להשתמש בה כדי לשחרר action, שאותו ה-reducer תופס. ה-type action נראה כך:

```
{ type: 'ADD_ITEM', payload: 'Item #1' }
```

ובן שהtekst משתנה בהתאם למה שנרצה. במקרה זה, מי שיפעיל את הפונקציה יוכל להעביר שם ארגומנט כרצונו.

מה שיש פה עוד – אף על פי שפונקציונליות זה לא נדרש – הוא export לשם פעולה שהוא באוטוות גדולות. מי שישתמש בשם action הוא גם ה-reducer. מקובל לעשות את זה כדי למנוע בעיות של שגיאות כתיב.

ה-reducer תופס את ה-type action וצריך ליצור אותו. ניצור תיוקית reducer ושם ניצור קובץ בשם itemsReducers.js. התוכן שלו יהיה:

```

import { ADD_ITEM } from '../actions';
function manageList(state = { items: [] }, action) {
  switch (action.type) {
    case ADD_ITEM:
      const oldItems = state.items || [];
      return {
        ...state,
        items: oldItems.concat(action.payload),
      };
    default:
      return state;
  }
}
export default manageList;

```

ה-reducer זהה יעבד בכל פעם שנוצר action. אם יש action עם השם ADD_ITEM, הוא מכניס את text שהוא אליו אל מערך items, יוצר אובייקט חדש ומחזיר אותו. כבר אמרנו קודם שיצירת האובייקט החדש היא קריטית לאופטימיזציה של RIDKOS. אנו לא יכולים להחזיר את אותו state.

כדי שRIDKOS ידע על ה-reducer זהה, אנו צריכים לחסוף אותו לעולם. באותה תקופה של reducer ניצור קובץ `jsx/index` בו נכניס את התוכן הבא:

```

import { combineReducers } from 'redux'
import items from './itemsReducer'

export default combineReducers({
  items,
})

```

אנו משתמשים כאן בפקודה שנקראת `combineReducers` כדי לשדר את ה-`reducer` שלנו לחץ בסטייט הקשור אליו. במקרה הזה החלטתי שהחלק של הרשימה בסטייט הגלובלי יקרא `items`, כלומר שהסטייט הגלובלי יראה כך:

```
state.items = ['item #1', 'item #2', 'item #3'];
```

זהו, עד כאן ה-`setup`, וזה עבודה קשה. אבל מהרגע זהה אפשר להשתמש ברידקוס בקומפוננטות שיש להן גישה לסטטייט. קומפוננטה שצרכיה לשנות את הסטייט מקבל, דרך ה-`sProps` שלה, את ה-`action` שמשנה את הסטייט, שאוטו כבר כתבנו.

קומפוננטה שצרכיה לקבל מידע מהסתטייט מקבל, דרך ה-`sProps` שלה, את החלק בסטייט שהוא צריך לקבל, וגם רידקוס תרנדר מחדש את הקומפוננטה הזו בכל פעם שהסתטייט הספציפי הזה ישתנה.

נתחיל בקומפוננטה שצרכיה לשנות את הסטייט. אני ארצה להעביר אל ה-`sProps` את ה-`action` המתאים: `addList`. איך אני עושה את זה? באמצעות `connector` שמחבר את הקומפוננטה הזו לרידקוס ומעביר, אל ה-`sProps` שלה, את ה-`action` שהוא צריך.

xs.js אמורה להיראות כך:

```

import React, { useState } from 'react';
import { TextField, Button } from '@material-ui/core';
import { addItem } from './redux/actions';
import { connect } from 'react-redux';
const mapDispatchToProps = (dispatch, ownProps) => ({
  addToList: (item) => dispatch(addItem(item)),
});
function ListInput(props) {
  const [item, setItem] = useState('');
  function changeHandler(e) {
    setItem(e.target.value);
  }
  function clickHandler(e) {
    props.addToList(item);
  }
  return (
    <div>
      <TextField onChange={changeHandler} />
      <Button onClick={clickHandler}>Insert item to list</Button>
    </div>
  );
}
export default connect(null, mapDispatchToProps)(ListInput);

```

הינה הצעדים שעשיתי כדי להמיר את `xs.js` לקומפוננטה רידקסטית:

צעד ראשון: ייבוא

```

import { addItem } from './redux/actions';
import { connect } from 'react-redux';

```

לייבא את החלקים של רידקסט (טוב, זה קל). אחד מהם הוא ה-`action` וחלק נוסף הוא פונקציית `connect`, שנדרשת באופן מפתחי.

צעד שני: למפות בין ה-`action` שאנו צריך ל-`props`

```
const mapDispatchToProps = (dispatch) => ({
```

```
    addToList: (item) => dispatch(addItem(item))
  })
}

הfonkzia haChosoba biYotar – ani be'atzm yozr fonkzia shmekblat argoment dispatch v'machzirha avo'iket.
```

```
{
  addToList: function(item) {
    return dispatch(addItem(item));
  }
}
```

כלומר, הפונקציה הזו דואגת לשים את props.addToList ב-props של הקומפוננטה. אם מישחו יפעיל את props.addToList, האירוע addItem יופעל. הפרמטר item הוא קריטי כי הוא בעצם הטקסט שמעבר.

הפונקציה הזו היא אחת המבלבלות ברידקס, אבל צריך לזכור שהיא בעצם הקשר בין ה-action ל-.props

צעד שלישי: להפעיל את ה-action שיש לנו ב-props איפה שצרי

```
function clickHandler(e) {
  props.addToList(item); // calling Redux
}
```

בקומפוננטה יש לנו רק מקום אחד כזה – כאשר אניلوحץ על כפתור ה"הוספה". מה שיש בסטייט הפנימי, הלוא הוא item, משוגר אל ה-action ומשם אל הסטייט הגלובלי. ה-action נמצא ב-props והובא לשם על ידי פונקציית ההיבור.

צעד רביעי: ליצא את הקומפוננטה באמצעות connect

```
export default connect(
  null,
  mapDispatchToProps
)(ListInput)
```

על מנת שרידקס ידע על כל מה שעשינו, אנו לא מייצאים את הקומפוננטה באמצעות connect כרגע, אלא עטופה במתודת connect. הארגומנט הראשון שמור לפונקציית מיפוי בין ה-props

לסטיט. במקרה זהה הקומפוננטה לא משתמשת בסטייט הגלובלי (רק משנה אותו, לא זורכת ממנו) ולפיכך הוא יהיה `null`. הארגומנט השני הוא המיפוי בין ה-`actions` לפרופס שאותו הסבירנו בסעיף הקודם. לבסוף – אנו מעבירים את הקומפוננטה עצמה. מהנקודה זו לא צריך לעשות כלום. כל מה שתכניסו בשדה ה-`key` יכנס לסטיט הגלובלי. אתם יכולים להכניס `console.log` כדי לראות שהכל עובד כמו שצריך.

עכשיו נמשיך אל הקומפוננטה שמציגה את הסטייט, הלווא היא `jsx.ListView`. כך היא נראהת כאשר היא מחוברת לRIDKOS:

```
import React from 'react';
import { List, ListItemText } from '@material-ui/core';
import { connect } from 'react-redux';
const mapStateToProps = (state) => {
  return state.items || [];
};
function ListView(props) {
  console.log('props.items', props.items);
  const listItems = props.items || [];
  const listItemsJSX = listItems.map((item) => (
    <ListItemText key={item}>{item}</ListItemText>
  ));
  return <List>{listItemsJSX}</List>;
}
export default connect(mapStateToProps, null)(ListView);
```

אליה הצעדים שעשיתי על מנת לחבר אותה:

צעד ראשון: `import`

```
import { connect } from 'react-redux';
```

טוב, זה הצעד הכי קל. אנחנו פשוט זקוקים ל-`connect`.

צעד שני: מיפוי

```
const mapStateToProps = (state) => {
  return state.items || [];
};
```

אני בעצם אומר שהיא תהיה ב-`props.items` הוא הסטיט הגלובלי->`items` – זו פונקציית המיפוי.

צעד שלישי: ליצא את הקומפוננטה באמצעות `connect`

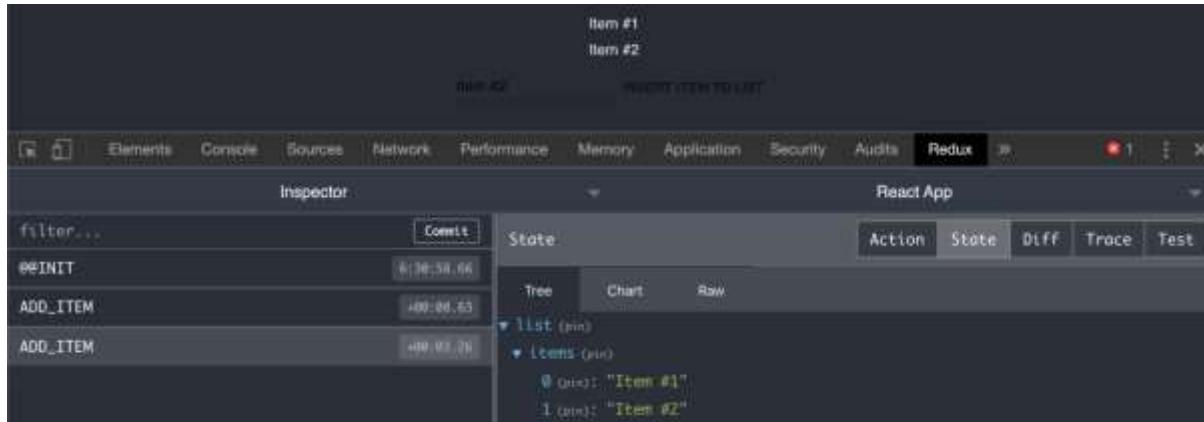
```
export default connect(
  mapStateToProps,
  null
)(ListView);
```

כמו בקומפוננטה שכתבה לסטיט, גם הקומפוננטה הזו צריכה להיות מוצאת באמצעות `connect` שמקבל ארבעה ארגומנטים. הראשון הוא המיפוי בין ה-`props` לסטיט הגלובלי. השני הוא המיפוי בין ה-`props` ל-`actions`. כיוון שהקומפוננטה היא רק קוראת ולא כתבת לסטיט הגלובלי, אני עבור שם `null`. וכמו כן, `ListView` הוא הקומפוננטה.

אבל, יתכן בהחלט מצב שבו יש קומפוננטה שגמ קוראת מהסטיט הגלובלי וגם כתבת לו, אז אנו נעביר בפונקציית ה-`connect` שתי פונקציות מיפוי ולא יהיה שם `null`.

אחרי שתעשו את כל זה, תוכל לראות שהאפליקציה קבוצה באמת עובדת ואפשר להוסיף להויסף לה איטמים בראשימה. זו עבודה מטורפת לאפליקציה כל כך קטנה, אך אפליקציות מורכבות יותר מאוד יהנו מרידקן.

כדי לנצל במאה אחז את היכולות של רידקס, כדאי להתokin את כל המפתחים של רידקס, שזמן גם בפיירפוקס וגם בכורום. יש לחפש DevTools Redux בחנות התוספים ולהוריד אותו בחינם ובמהירות. לאחר ההתקינה תראו לשונית של רידקס בכל המפתחים. לחיצה עליה תראה לכם את הדיבאגר של רידקס. בדיבאגר אפשר לבדוק את כל מה שקרה בסטיט הגלובלי ולבוחן היבט כל `action` ו-`reducers`.



אפשר לנوع קדימה ואחורה בזמן ולראות מה השתנה. ממשך המשתמש של כל המפתחים פשוט ויכול לנסות אותו.

ברידקס יש הרבה יותר מניהול סטטי. יש לנו גם MiddleWares, שיכולים ליצור עוד תהליכיים, נוספים על שינוי הסטטי, ולסייע לנו בניהול ובתפעול מתקדם של האתר ושל האפליקציה. לימוד רידקס הוא מטלה קשה ואפיילו מסובכת, ויצא לה (בצדק או שלא בצדק) שם של שהוא שקשה לعشות, אך בהחלט שווה להתאמץ כיון שנכון להיום – כל אפליקציה גדולה ואטר גדור משתמשים בכל ניהול סטיטיים, ורידקס היא אחד הכלים הפופולריים.

תרגיל:

הויספו כפטור איפוס רשיימה כרכיב נפרד ששמו הוא `ResetButton` וימוקם ב-`Container.jsx`.

פתרונות:

ראשית ניצור את ה-`action`.`action` יגרום לרשיימה להתאפס. בקובץ `js/index.js` בתיקיית `actions` נוסיף את הפעולה של `RESET_LIST`:

```
/*
 * action types
 */

export const ADD_ITEM = 'ADD_ITEM';
export const RESET_LIST = 'RESET_LIST';

/*
 * action creators
 */

export function addItem(text) {
  return { type: ADD_ITEM, payload: text }
}

export function resetList() {
  return { type: RESET_LIST }
}
```

שימוש לב: בשונה מה-`action` הראשון, ל-`action` של איפוס יש רק `type` ואין `Payload`.

הצעד השני הוא להוסיף reducer, או נכון יותר – להוסיף ל-reducer הקיימן יכולת לטפל ב-action החדש שלנו, שהוא ניקוי הרשימה:

```
import { ADD_ITEM, RESET_LIST } from '../actions';
function manageList(state = { items: [] }, action) {
  switch (action.type) {
    case ADD_ITEM:
      const oldItems = state.items || [];
      return {
        ...state,
        items: oldItems.concat(action.payload),
      };
    case RESET_LIST:
      return { ...state, items: [] };

    default:
      return state;
  }
}
export default manageList;
```

הקוד פשוט הופך את ה-items לאובייקט ריק ומשכפל אותו הלאה. להזכירם, אנו חיבים להחזיר אובייקט חדש כיון שכך המנווע של RIDKO ידע לרנדר את הקומפוננטה. הייצור של האובייקט החדש נעשה כר:

```
return { item: [...state, []] };
```

הצעד השלישי הוא ליצור את קומפוננטת Button ול לחבר אותה לRIDKO ולפעולת LIST שהפונקציה resetList יורה.

নির্মাণ ফনকেশন :ResetButton.jsx

```

import React from 'react';
import { Button } from '@material-ui/core';
import { resetList } from './redux/actions'
import { connect } from 'react-redux'

const mapDispatchToProps = (dispatch) => ({
  resetList: () => dispatch(resetList()),
});

function ResetButton(props) {

  function clickHandler(e) {
    props.resetList(); // calling Redux
  };

  return (
    <div>
      <Button onClick={clickHandler}>Clear List</Button>
    </div>
  );
}

export default connect(
  null,
  mapDispatchToProps
)(ResetButton);

```

החלק המשמעותי הוא המיפוי שמכניס את ה-action אל ה-props:

```

const mapDispatchToProps = (dispatch) => ({
  resetList: () => dispatch(resetList()),
});

```

ה-`resetList` נכנס לתוך ה-`props` והפעלה שלו תפעיל את `dispatch(resetList)`, שזו הפעולה. וכמו כן נחבר את `mapDispatchToProps` באמצעות `:connect`

```
export default connect(
  null,
  mapDispatchToProps
)(ResetButton);
```

הפעולה الأخيرة שיש לעשות היא להוסיף את רכיב ה-`ResetButton` אל ה-`Container.jsx`.

```
import React from 'react';
import ListInput from './ListInput';
import ListView from './ListView';
import ResetButton from './ResetButton';

function Container() {
  return (
    <div>
      <ListView />
      <ListInput />
      <ResetButton />
    </div>
  );
}

export default Container;
```

אפשר לראות איך, לאחר שכבר עבדנו קשה להוסיף את רידקס לאפליקציה שלנו, הוספה של פעולות הופכת לקלת מואוד.

העבודה הקשה ברידקס זה ליצור את המבנה של רידקס עצמו. לאחר העבודה הזאת, כל מה שנותר לנו לעשות זה לקשר את הקומפוננטות השונות – כל אחת לפי תפקידיה: אלו שקוראות יkoshrו באמצעות `mapStateToProps` ו-`connect` ואלו שכותבו יkoshrו באמצעות `mapDispatchToProps` ו-`connect`. כאשר ניתן גם לקשר קומפוננטה גם לכתיבה וגם לקריאה.

תרגיל:

הוסיפו קומפוננטה נוספת המציגת את הרשימה אופקית במקביל לרשימה האנכית. שם הקומפוננטה הוא `HorizontalList`.

פתרון:

אנו ניצור את קומפוננטת `HorizontalList.jsx`. היא תהיה קומפוננטה פשוטה למדי, אך היא תהיה מחוברת לסטיבט הגלובלי באמצעות `connect` כדי לבדוק כמו קומפוננטת הרשימה הקיימת.

```
import React from 'react';
import { connect } from 'react-redux';
const mapStateToProps = (state) => {
  return state.items || [];
};

function HorizontalList(props) {
  console.log('props.items', props.items);
  const listItems = props.items || [];
  const listItemsJSX = listItems.map((item) => <span
key={item}>{item},</span>);
  return <div>{listItemsJSX}</div>;
}
export default connect(mapStateToProps, null)(HorizontalList);
```

על מנת לראות איך אפליקציית הרידקס עבדת ממש בחן, וגם לשנות בה דברים, ניתן להכנס אל הקישור הבא:

<https://codesandbox.io/s/redux-app-example-eigl1>

פרק 25

סיום - ומה ענשין?



օיּוּם – וְמָה עֲכַשֵּׂו?

אם קראתם את כל פרקי הספר, אתם יודעים לבנות אפליקציות ריאקטיביות. אבל הידע שלכם הוא עדין תיאורטי, ולימוד פרימיום חדש נעשה באמצעות הידים. אי-אפשר ללמד פרימיום רק אם לא עברתם את השלבים שאף ספר לא יכול להעביר אתכם: התסכול המתלווה למשהו שלא עובד כמורה, השמחה הגדולה כאשר מצליחים לפתרו את התקלה ורוכשים ידע נוסף יקר מפז, החיפוש המתמיד בגוגל וב-stackoverflow ובנייה מוצרים אמייתים ופתרון התקלות שיש בהם.

אם סיימתם לקרוא את הספר ולפתרו את התרגילים, הצעד הבא שלכם הוא לבנות אתר, אפליקציה או יישום מבוססי ריאקט. זה הזמן להצטרף ל专家组 שavanaugh אתרים או אפליקציות בהתקנות עבור מיזמים הקרובים ליליכם, לבנות אתר או אפליקציה לחבר שתמיד היה לו רעיון או לעמותה שתמיד רציתם לתרום לה. הניסיון המעשי הראשון של בניהית אתרים ובכתיבת קוד היה עבור עמותת טולקן (מחבר "שר הטבעות" בישראל – אין סיבה שההvr לא יהיהvr עבורה). הניסיון כאן הוא קרייטי ואתם חיבים לבנות משהו משל עצמכם, אפילו ברמת הניסיון. אסור להסתפק בתרגילים שיש בספר.

כאשר אתם מחליטים לבנות משהו שלכם – מילת המפתח היא תכנון. אחד ממנהיגי הפיתוח שלי נהג לומר ששבועיים של תוכנות חוסכים שעתיים של תכנון. חשבו על הקומפוננטות, בדקו באילו קומפוננטות חיציות אפשר להשתמש ואיך הן עובדות ואם אפשר להשתמש בהן עבור האתר שלכם. נוסף עלvr כר, בקשו לקבל ביקורת על העבודה שלכם וcrc תגלוvr איךvr יכוליםvr לעשותvr אותהvr טובvr יותר.

אבל אחרי שבניתם את הפרויקט שלכם – יש עוד כמה צעדים שתצטרכו לעשות כדי להשתלב בשוק העבודה. אף אחד לא יכול אתכם לעבודה על סמך קריית הספר הזה ופרויקט גמר, מצליח ככל שיהה. יש כמה נתיבים נוספים שיאפשרו לכם להמשיך ללמידה ולצבור ניסיון – עוד לפני העבודה הראשונה.

התחברות לקהילת הפיתוח

אחרי שבניתם את הפרויקט שלכם, או במהלך הפיתוח שלו – חשוב גם להצטרף לקהילת הפיתוח הישראלית. היא חמה ונעימה ויש לא מעט מתכנתים שיוכלו לשיער לכם להמשיך הלאה. בפייסבוק יש קבוצה פעללה מאוד של מתכנת ריאקט שפועלת בעברית וכדי להצטרף אליה ולהשתתף (בענוה הרואיה) בדיונים, לקרוא, ללמידה ולהבין לאן כדאי להתקדם, ואם יש פרויקט פתוח שמחכה למתרנדבים – נסו להצטרף אליו.

פגשים ומיטאפים

באטר meetup.com יש לא מעט מידע על מפגשים בחינוך שנערכים מיטאפים. במפגשים הללו נפגשים מתכנתים וشומעים הרצאות או בונים יחד דברים. יש מפגשים המיעדים למתכנת ריאקט וכדי להירשם לקבוצות השונות שמארגנות מפגשים על מנת לדעת על אלו שמתעניינים בסביבתכם

ובשעות המתאימות לכם. במפגשים הללו אפשר גם להמשיך ללמידה על פיצ'רים שונים בריект ועם להיפגש עם מתכנתים מוכרים, לשאול לעצם ולקבל הכוונה בנוגע לתקומות לימודי השפה ובפרויקטים אחרים. מדי פעמיים יש באתר הזמן לאקטים ולמפגשים שבהם מתכנתים בפועל. כדי מאד להציג לך קבוצה צו וلتכנת לצד מתכנתים אמיתיים בליווי הכוונה אמיתית. כמה שעות עבודה כלו יקדמו אתכם באופן מטאורי, גם אם לא תצכו בפרס.

האתר Stackoverflow

האתר Stackoverflow הוא שימושי בשתי דרכים – ראשית הוא מכיל תשובה 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 מתכנתים אחרים, שואלי פחות מנוסים מכם בריקט בתחום התיאורטי. השתתפות בדיון וניסיון לענות על שאלות של אחרים בריקט גם יקדמו מאד את הידע שלכם. פרופיל עשיר ב-Stackoverflow הוא משזה שראוי לציין בקורות החימם.

תרומת קוד בגייטהאב

לבסוף, אפשר לתרום קוד בגייטהאב לפרויקטם שמשתמשים בריקט. למשל, ספריית קומפוננטות SMBusKit על ריאקט. בגייטהאב יש לא מעט פרויקטים שימושיים לתרומת קוד. במהלך הפרויקט שבניהם בודאי ראייתם ספריות קומפוננטות שחררה להן בדיק הקומפוננטה זו שהייתם רוצים. למה שלא תוסיףו אותה? או תתקנו את הדוקומנטציה? או תכתבו בדיקה אוטומטית במקום שבו היא חסורה – תרומת הקוד הזה היא ממש בבחינת ניסיון, ואם תבחרו בנתיב זהה ותצלחו בו, תוכלו אפילו להשתלב בקהלות בתעשייה התוכנה.

מצד שני, העבודה בגייטהאב מצריכה ידע בגייט, הבנה באנגלית ויכולת לתקשר עם המתכנתים שעובדים בפרויקט ומגנים מכל העולם. יש גם מיטאים שבהם מתכנתים אחרים מסבירים איך עובדים עם גיטהאב ותרומים קוד.

כך או כך – הספר הזה הואצעד הראשון בכניסה לעולם המרתך של ריאקט ופיתוח צד לקוח. 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 שמשתמש (או, במקרה מסוים יותר, צריך) את הקומפוננטה שלנו לא משתמש ב-props המתאים? הקומפוננטה לא תעבור או שתעביר באופן לא צפוי. אם אנו כתבים קומפוננטה, החובה שלנו היא לסמן לאלו שימושים בה (גמ אם המשתמשים בה הם אנחנו, בעתיד) באיזה props אפשר להשתמש. כדי شيء שימוש בקומפוננטה יקבל חוויה מיידי. איך עושים זאת? באמצעות PropTypes. מודול שהוא חלק מריאקט עד גרסה 15.5 והחל מהגרסת הzo הוא יצא למודול משלהו שניינט להתקינו בנפרד. PropTypes הוא הדרך שלנו להציג איזה props אנו נדרשים לקבל ומאייה סוג. כך למשל, אני יכול להציג על props מסוימים שנדרשים ועל כלו שהם רק אופציונליים וכמו כן לפרט את הסוגים שלהם. כך, אם אמי שימוש בקומפוננטה שוכן להעיבר props מסוימים, הוא יקבל התראה. אם הוא מעביר מחזרות טקסט בעוד אנו מצפים למספר, הוא מקבל גם כן התראה מאוד ברורה.

למרות ש-PropTypes אינם חלק מריאקט, רבים מהמתכנתים שפתחים בריאקט עובדים איתם ומומלץ מאוד לעשות כן.

התקנת PropTypes

בדוק כמו ספריית קומפוננטות חיצונית, שלמדנו עלייה בעבר, או Create-React-App, או מתקנים את PropTypes באמצעות `npm install prop-types`. להתקנת מודולים: `node-sjs`. בהנחה ש-`node-sjs` מותקנת במחשבכם ואתם משתמשים באפליקציה `Create-React-App`, הכנסו באמצעות הטרמינל לתיקייה הראשית של האפליקציה שלכם והקלו את הפקודה הבאה:

```
npm install prop-types
```

אם הכל תקין, לאחר דקקה תוכלו לראות בטרמינל חוויה על כך שההתקנה הצלחה. מהנוקדה הzo תוכלו להשתמש ב-PropTypes.

```
* my-app git:(master) ✘ npm install prop-types
[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-pnp@1.1.2 requires a peer of typescript@* but none is installed. You must install peer dependencies yourself.]
[NPM 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 פונקציה. אנו מיבאים את PropTypes באמצעות import, בדיקן כמו כל ספריה חיצונית אחרת שלמדנו עליה ובאמצעות האובייקט שאנו מיבאים אנו מצהירים על ה-props השונים.

הבה ונדגים באמצעות הקומפוננטה פשוטה זו:

```
import React from 'react';

function Welcome(props) {

  return (
    <p>Hello, {props.name} !</p>
  );
}

export default Welcome;
```

אם אנו רוצים להשתמש בקומפוננטה זו, אנו נעשה זאת, למשל, כך:

```
import React from 'react';
import './App.css';
import Welcome from './Welcome';

function App() {
  const userName = '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 ה-props המתאים? למשל אם אני לא מעביר name? זו תקלה שיכולה להתקיים. מהשלב שבו כתבתי את קומפוננטת Welcome עד השלב שאני משתמש בה יכולם לעבור חודשים, אולי שנים. אם אשכח את מה שאני אמר לנושא - מה שיקרה הוא שהקומפוננטה לא תעבוד כמורה ולי לא יהיה מושג למה. אני אctrיך לניבור בקוד כדי להבין מה השتبש.

הבה ונshall use PropType. ראשית יבוא אז הצהרה:

```
import PropTypes from 'prop-types';
import React from 'react';

function Welcome(props) {

  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 שאני 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 **drop**. חלק שלישי הוא האם מדבר ב-**drop** 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-`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 עבור סוג נתון שאינו תואם לסוג שאני מבקש, אני מקבל גם הודעה שגיאה ברורה שמודיעה לי ש-`name` מקבל 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 כרגע ותנסה לנדר את הפלט. אם לא יהיה 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 הפרימיטיביים האפשריים. בטבלה זו יש ריכוז של הסוגים הנפוצים:

סוג הערך	תיאור	דוגמה
מחרוזת טקסט	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>
fonckzieh	כל פונקציה, רגילה או אונונימית	<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	קומponeנטה ריאקטיבית כלשהי שמועברת כפרמטר	<code>PropTypes.element</code>

אני יכול להחליט שה-`PropType` שלי יקבל כמה סוגים. כך למשל, אם הקומponeנטה שלי יודעת להתמודד עם מספר או טקסט ב-`prop` מסוים. אני יכול לציין שאני יכול או סוג אחד, או סוג שני. אני עושה את זה באמצעות הפקודה `elementTypeOfType` במקום הסוג.

למשל:

```
import PropTypes from 'prop-types';
import React from 'react';

function Welcome(props) {

  return (
    <p>Hello, {props.name} !</p>
  );
}

Welcome.propTypes = {
  name: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number,
  ]).isRequired,
}

export default Welcome;
```

אני תמיד יכול לשרשר את `isRequired` במידה והפרמטר נדרש. אך זו לא חובה.

חשוב להזכיר `PropTypes`. מזכיר בתוספת קטנה וחשיבותה מאוד לכל קומפוננטה שיכולה לעשות סדר בקוויד. נכון, זה מעט יותר עבודה, אבל לטווח ארוך זה חשוב וצדאי.

נספח: שינויים מהמהדורה הקודמת (מהדורה 1.0.0)

פרק הרידקס הוחלף לחליות בדוגמה פשוטה יותר. בדוגמה החדשה אין אובייקט `list` שבתוכו מערך של `items` אלא מערך של `item` בלבד. בנוסף, הוכנה אפליקציית ריאקט ב-`so.sandbox` וקישור אליה הוצב בסוף הפרק.

נוסף תרגיל לפרק הרידקס.

הדוגמה בפרק על הסטייט הוחלפה לדוגמה הכללת אירוע ולחיצה במקום דוגמת `setInterval`.

תיקוני שגיאות הקלדה בפרק אודוט המרצים ואודוט `Honeybook`.

עדכון הדוגמאות והקישורים בפרק הפתיחה לריאקט גרסה 17.

תודה רבה על הרכישה של הספר!

עבדתי מאוד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוצריו העריכה. יותר מ-1800 אנשים תמכו בספר זהה ואייפשרו 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. מתווך תקווה שהרוכש וה透מך לא ינצל את האמון שנותני בו להעתקה סיטונאית של הספר לאנשים אחרים והפצה שלו. אני מאמין שרוב האנשים הוגנים.

העתיק זהה נמכר ל:

nivbarsh@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נחברים פרט' הרוכש באופן שקוֹף למשתמש. כדי מאד להמנע מהעתיקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיחקנו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!