

תודה רבה על הרכישה של הספר!

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

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

העתק זהה נמכר ל:

eli.k23@gmail.com

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

תודה וקריאה נעימה!

ללמוד ריאקט בנברית

רן בר-זיק

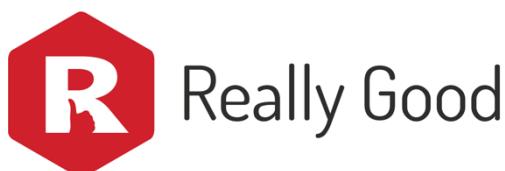


הקריה האקדמית אונו
Ono Academic College
החוג למדעי המחשב

לימוד ריאקט בעברית

רן בר-זיק

מהדורה: 1.0.0



כל הזכויות שמורות © רן בר-זיק, 2020.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רישיון לא-בלודי, לא-ייחודי, אישי, בלתמי נתן להעbara (למעט על פי דין), ובلتמי 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 לצלט קטעים קצרים מהספר במסגרת הגנת שימוש הוגן, ככלומר פסקה או שתים, כאשר אתה מפנה למקור ומציר את רן בר-זיק כמחבר הספר.

הדוגמאות המובאות בספר זה הן בבעלות של רן בר-זיק, ואסור לך להשתמש בהן בתחום תוכנות שתפתח. אם אתה רוצה להכנס אונטן לפרויקט שלנו, 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

על "לימוד ריאקט בעברית"	10
על המחבר	11
על העורכים הטכניים	12
דoron זבלסקי	12
gil פינק	12
דורון קילוי	13
על החברות התומכות	14
Really Good	14
אלמנטור	14
HoneyBook	16
על ריאקט	17
דרכ הليمוד	19
על המונחים בעברית	19
סביבת העבודה הבסיסית	21
סביבת עבודה בסיסית ב-codepen	24
אפליקציית ריאקט	27
בנייה קומפוננטת פונקצייה בסביבת העבודה הבסיסית	33
בנייה של סביבת ריאקט אמיתית ומורכבת יותר באמצעות Create React App	42
התקנת Node.js על המחשב שלכם	42
התקנה על חלונות	43
התקנה על מac	46
התקנה על לינוקס	47
עבודה עם טרמינל	47
הפעלת הטרמינל	47
ביעוט בטרמינל	49
מציאת מקומות בטרמינל דרך חלונות	50
טרמינל ב-Node.js	51
Visual Studio Code	51
בדיקות גרסה Node.js דרך הטרמינל	
Create React App	52
עבודה עם Create React App	52
בעיות ותקלות	57

59	כתיבת קומפוננטה ראשונה ב-React App Create
68	Export \ Import
71	יבוא מנתיבים אחרים
73	אין חובה להשתמש בסימט הקובץ js
73	"יצוא של כמה משתנים
74	"יבוא של תמונות, CSS ומשאים אחרים
77	JSX
84	רשימות ב-JSX
93	קומפוננטה עם תכונות (<i>props</i>)
107	דיבאג
112	סמיין
114	ריאקט ורנדור
115	סטייט
124	תכנון מבנה הקומפוננטות
129	איורים ועדרון קומפוננטות
130	אירוע DOM
131	איורים סינטטיים / ריאקטיים
152	אלמנט <i>fragment</i>
155	useEffects
161	קומפוננטה ללא שם
164	עיצוב קומפוננטות
166	CSS בסיסי
167	סלקטו
169	תובנות
173	קלאס ריאקטוי
183	מעגל החיים בклאס
183	componentDidMount
183	componentDidUpdate (prevProps, prevState, snapshot)
183	componentWillUnmount
183	shouldComponentUpdate (nextProps, nextState)
184	getDerivedStateFromError (error)
184	componentDidCatch (error, info)

192.....	שימוש בקומפוננטות מקורות אחרים.....
197.....	יבוא כמו קומפוננטות.....
199.....	מודולים חסריים.....
212.....	HOC: Higher Order Component
226.....	ראונינג.....
249.....	קונטנסט.....
262.....	חיבור לשרת עם סרוויסים.....
270.....	מבוא לבדיקות עם jest.....
272.....	חלק ראשון – import.....
272.....	חלק שני – בתיבת מסגרת הבדיקה.....
272.....	חלק שלישי – הרצת הקומפוננטה.....
272.....	חלק רביעי – הבדיקה.....
275.....	בדיקות לקומפוננטה עם props.....
276.....	פונקציית בדיקה – assert.....
276.....	not.....
276.....	toBeTruthy.....
277.....	toBeFalsy.....
277.....	toContain.....
282.....	יצירת בילד והעלאת האפליקציה לserveה חייה.....
288.....	רידקס.....
289.....	הפילוסופיה מאחורי רידקס.....
290.....	קומפוננטות מנהלות מול מנהלות.....
296.....	הטמעת רידקס.....
296.....	התקנת רידקס.....
296.....	תכנון הסטייט וזרימת המידע במערכת.....
298.....	mapStateToProps.....
299.....	mapDispatchToProps.....
299.....	הימוש של רידקס במערכת שלנו.....
314.....	סיכום – ומה עכשוו?.....
315.....	התחברות לקהילת הפיתוח.....
315.....	מפגשים ומיטאפים.....
315.....	האתר Stackoverflow.....

316.....	תרומת קוד בGITהאב
317.....	נספח: <i>PropType</i>
317.....	התקנת PropTypes
318.....	השימוש ב-PropTypes
322.....	ערכים שאפשר לקבוע

על "לימוד ריאקט בעברית"

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

הספר מניח ידע מקיף בג'אווהסקרייפט וידע בסיסי ב-HTML וב-CSS. אם איןכם מכירים ג'אווהסקרייפט, אפשר למודד זאת בספר "לימוד ג'אווהסקרייפט בעברית". יש שם פרק המסביר על HTML בסיסי ועל CSS בסיסי. מי שסימן לקרוא את הספר ההוא ותרgal כהלה, אמרו להזיק במידע מספק על מנת להתחיל ללמד ריאקט. בספר זה נלמד על ריאקט ממש מהבסיס – הכרת מונחים בסיסיים ויצירת סביבת עבודה – ונגיע עד חומרים מתקדמים כמו `high order state hooks` ו-components. הספר מעודכן לגרסה ריאקט 16.

על המחבר

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

נוסף 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 והסבירים על תכונות בעברית ומתקדם לפחות פעם בשבוע.

REN הוא מחבר הספר "לימוד ג'אווה סкриיפט בעברית" והספר "לימוד Node.js בעברית" ומלמד בקורס האקדמית אוננו.

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

על העורכים הטכניים

דורון זבלסקי

הקריירה של דורון התחילה דואקן בצד השרת, בחברות אבטחת מידע שבהן עבד כמתנדס 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 ואמץ אותה בחום. דורון הקים את קהילת ריאקט בישראל, כולל קבוצת פיסבוק ומיטאפ פופולרי, ואף יוזם את כנס ReactNext הראשון בישראל והוא מופיע משותף שלו. בנוסף לכך הוא מרצה על ריאקט, על בדיקות אוטומטיות ועל בניית צוותים מנכחים ותורם מזמין בשמחה למתחילים המבקשים סיוע בתחום במסגרת מפגשים אישיים וקובוצתיים.

gil Fink

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

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

דורון קילזי

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

על החברות התומכות

Really Good

Really Good היא בוטיק פיתוח Front End שעובדת עם סטארטאפים וחברות טכנולוגיות מאז הקמתה ב-2012 על ידי שחר טל ורוני אורבך. אנחנו נוהנים לבנות אפליקציות מורכבות עם UX מוקפד במגוון טכנולוגיות ללקוחות מעוניינים שחוויות המשמש חשוב להם, ושומרים על איזון בריא בין עבודתה 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 שימושה את הדרך בה בונים אתרי אינטרנט בשוק המkteבי. אלמנטור מעניק למשתמשים את החופש ליצור עמודי אינטרנט ללא צורך בקוד ולפתחים את החירות לדחוף את הגבולות, לרענן ולהרחיב את המערכת בצורה קלה ומהירה באמצעות API ייחודי למפתחים, ובכך לחסוך זמן פיתוח ולהוות יעילים ורוחניים.

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

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

הנדסי DevOps ו Big Data Stack עם מומחיות בניהול Kubernetes על פלטפורמות הענן של AWS & GCP.

אתר החברה: <https://elementor.com>

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

HoneyBook

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

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

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

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

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

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

על ריאקט

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

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

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

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

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

המבנה של ריאקט ודרך העבודה בה לא שונות מהותית מספריות אחרות כמו אנגולר או Vue. אך אם איןכם מכירים אף פרימורק או ספריה של ג'אווה סקריפט, ריאקט היא מקום מצוין להתחיל בו את המסע לעולם המופלא של פיתוח צד לקוח. זאת אף שריקט שונה בכמה דרכים מהותיות, שאוון נלמד בהמשך, מספריות אחרות כמו אנגולר.

דרך הלימוד

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

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

על המונחים בעברית

אני כותב בעברית על טכנולוגיה ועל תכנות שכבר יותר מעשור, והדילמהabei לו מונחים בעברית להשתמש מלואה אותה תמיד. מצד אחד, האקדמיה ללשון העברית מספקת לנו מונחים רבים בעברית. מצד שני, בתעשייה ההייטק, שמןנה אני מגיע, איש לא משתמש ברבים מהמונחים האלה. אם תגינו לריאיון עבודה ותגידו: "במפגש המתכנתים האחרון שמעתי על דרך חדשה לבצע הידור שבודק הזחות במנשך מבוסס הבטחות", סביר להניח שלא תקבלו את העבודה. אבל אם תגידו, "במייטהפ האחרון שמעתי על דרך חדשה לבצע קמפול שבודק אינדנטציה-ב-API מבוסס פרומיסים" – יבינו על מה אתם מדברים. זו הסיבה שלא תמצאו בספר מילויים 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 להכניס אותו לחיפושים שלכם בגוגל.

חשיבותו של אקדמיה ללשון וschlüsse מהמוניים שלה אכן נכנסו לשפה המדוברת במרכז הטכנולוגיה השונים (למשל: קובץ או מסד נתונים), אבל בכל מקום שהיתה בידי הבחירה בין להיות מובן לבין לעמוד בכללי הלשון, העדפתו להיות מובן.

סביבת העבודה הבסיסית

זה הפרק החשוב ביותר, כיווןuai-אפשר למדוד קוד בלי ל לכלך את הידים 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 עבודה היא החלק הקשה ביותר בלימוד טכנולוגיה חדשה. נסו שוב ושוב – בצעו ריסטרט למחשב, נסו מחשב אחר, שדרגו את מערכת הפעלה, נסו מדף אחר שאינו הדפדפן הרגיל שלכם – כל טכניתה וטכניתה. פשוטαι-אפשר לדלג על השלב זהה.

ריאקט מורכבת מכמה חלקים – כולם בג'אווהסקרייפט. החלק הראשון הוא הספרייה עצמה: ריאקט. מדובר בקובץ ג'אווהסקרייפט מרכזי המכיל את כל הפקנציונליות של הספרייה. הקובץ מכיל מודולים של ג'אווהסקרייפט וגם משתנים גLOBליים ואחרים – ובולדיוαι-אפשר להשתמש בריאקט. כשאנו בונים סביבת עבודה בסיסית, אנו זקוקים לריאקט. הגיוני, לא? החלק השני הוא dom-react. גם הוא חלק מהספרייה וגם הוא כתוב בג'אווהסקרייפט. הוא מכיל את הפקנציות הקשורות בין ריאקט ל-DOM. ראשית התיבות של DOM הם Document Object Model, ואני ארכחיב לגביו בהמשך. כרגע צרייך פשוט לזכור שמדובר בעוד חלק של הפרימיום ורק שאנו צריכים איתנו כשהאנחנו מפתחים עבור אתר אינטרנט.

החלק השלישי הוא babel. מה זה? מדובר בספרית ג'אווהסקרייפט שימושית ופופולרית מאוד. יש לה כמה תפקידים חשובים. במקור babel סייעה למפתחים שרצו להתאים את קוד הג'אווהסקרייפט שלהם לדפנאים ינסים שלא תמכנו בפיצרים החדשניים של השפה, למשל דפנאים שלא ידעו מה זה let או const. הספרייה זו לקחה את כל הקוד המודרני של ג'אווהסקרייפט והעבירו אותו לתוך, שבמסגרתו הוא הפך לקוד שתואם גם דפנאים ינסים. למשל, היא המירה את let ל-var, כך שדף יישן יוכל לעבוד איתו. התהליך הזה נקרא "טרנספיירציה" – זו המילה המדעי המחשב שמשמעה לחתוך קוד שכתוב בשפה מסוימת ותרגמו אותו לקוד שכתוב בשפה אחרת. במקרה הזה לחתוך קוד שכתוב בג'אווהסקרייפט מודרנית ולהעביר אותו לקוד שכתוב בג'אווהסקרייפט מגרסאות קדומות.

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

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

אנו יכולים להוריד את הקבצים האלו מהאתר של ריאקט או להשתמש בהם **ישירות-cdn**. ראשית התיבות של CDN הם Content Delivery Network – זהינו לשרתים גדולים שנמצאים בכל מקום בעולם. חלק מהם מציעים שירות פתוח לציבור של אחסון קבצים של ספריות גדולות ומוגן שריאקט, הפופולרי בעולם, נמצאת בינהן. ה-CDN שהדוקומנטציה של ריאקט משתמש בו הוא `unpkg` וsono משתמש בו. `unpkg` מציעה את הקבצים בקישורים הבאים:

קובץ הפרימורק של ריאקט:

<https://cdnjs.cloudflare.com/ajax/libs/react/16.4.0/umd/react.development.js>
שימוש לב: הספר מלמד על גרסה 16 של ריאקט ולכן קיים המספר הזה ב-URL.

קובץ react-dom:

<https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.4.0/umd/react-dom.development.js>

קובץ babel:

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

כל קובץ ה-HTML שלנו נראה:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/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>
```

בתחתית הקובץ נמצא הקוד שלנו. אנו מסמנים לו babel שהוא קוד שציריך לעבור טרנספילציה באמצעות הצ마다 של:

`type="text/babel"`

לתניית `script`. בין התגיית הפותחת של `script` לתגיית הסגורה שלו, אנו יכולים ל כתוב את הקוד הבא:

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

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

סביבה עבודה בסיסית ב-codepen

אפשרות נוספת היא לבנות סביבת עבודה מרוחקת, שגם בה אפשר לכתוב ג'אווהסקריפט ולצפות בתוצאות באמצעות דףדף בלבד. סביבת העבודה זו זמינה בחינם בכמה וכמה כלים וארטירים, אבל האתר המכני פופולרי ומוצלח הוא so.iodepen. מדובר באתר שמאפשר לנולם (אפיו 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/>: נכנסתם אל

מתחלים לכתוב קוד ורואים את התוצאות.

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

Pen Settings

HTML CSS JavaScript Behavior Screenshot

JavaScript Preprocessor

Babel

Add script as a module

Add `type="module"` on the Pen's `<script>` tag

Add External Scripts/Pens

Any URL's added here will be added as `<script>`s in order, and run before the JavaScript in the editor. You can use the URL of any other Pen and it will include the JavaScript from that Pen.

Recent: **React DOM** React

- https://cdnjs.cloudflare.com/ajax/libs/react/16.4.0/umd/react.development.js
- https://cdnjs.cloudflare.com/ajax/libs/react-dom/16.4.0/umd/react-dom.development.js

Quick-add: ---

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

<https://codepen.io/barzik-the-vuer/pen/gOYMWoP>

.Here will be React code צפו בקונסולה וראו את המסר

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

פרק 1

אפליקציית ריאקט



אפליקציית ריאקט

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

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

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/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 שלנו. אנו צריכים רק לזכור שה-`id` הוא `.content`. את אפליקציית הריאקט אנו יוצרים באמצעות:

ReactDOM.render

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

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 נכון. אם לא הכל נכון וראים לא Hello World! הפסיקו לקרוא, פתחו את הקונסולה, צפו בשגיאות וחפשו אותן בגוגל כדי לתקן, ודאו שהעתקתם את הקוד כשורה והמשיכו רק כאשר אתם יודעים שסביבת הקוד שלכם עובדת.

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

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

אני מעביר אותו לקבוע target כארגומנט השני. אתם אמרוים להזכיר את זה אם יש לכם ידע בג'אוהסקרייפט. אם לא, אנא חזרו על החומר של ג'אוהסקרייפט ו-HTML (שנמצא גם בספר שלי "לימוד ג'אוהסקרייפט 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'אוהסקרייפט! איך זה יכול להיות? הרי אם תשתמשו בקוד HTML ללא מירכאות בקוד ג'אוהסקרייפט, הקוד ידפיס שגיאה ויפסיק לפעול. ג'אוהסקרייפט לא מכירה HTML ולא יודעת לעבד אותו. איך יכול להיות שאינו משתמש ב-`h1` וב-HTML בג'אוהסקרייפט ללא מירכאות והקוד לא נופל?

הסיבה היא JSX. זוכרים את המרכיב השלישי בסביבת הפיתוח שלנו, ה-babel? הוא אחראי למצוא את כל התגיות של HTML שיש בקוד הג'אוהסקרייפט ולהמיר אותן למשהו שנג'אוהסקרייפט יודעת להתמודד איתו. הקוד הזה, שכרגע יש בו HTML פשוט בלבד, הוא JSX – ראשית תיבות של JavaScript XML – 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'אוהסקרייפט 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.

תרגום

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

פתרונות

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/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 לאפליקציה ריאקט וועליהם הסבכנו בתחלת הפרויקט.

הצעד הנוסף הוא ליצור צפ שיש לו פג 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

הממודה זו מקבלת שני ארגומנטים: האחד הוא JSX והשני הוא האלמנט שהאפליקציה תהיה בו. הארגומנט הראשון הוא JSX פשוט שאינו שונה מ-HTML, מלבד העובדה הפשטת שהוא בתוכו קובץ ג'אווהסקריפט:

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

הารוגומנט השני הוא האלמנט שהאפליקציה תהיה בו. אני מקבל אותו באמצעות `getElementById`, שהוא מתודת ג'אווהסקריפט שנמצאת בדף באופן טבעי (בלי קשר לפרויקט), ו מעביר אותו באמצעות משתנה:

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

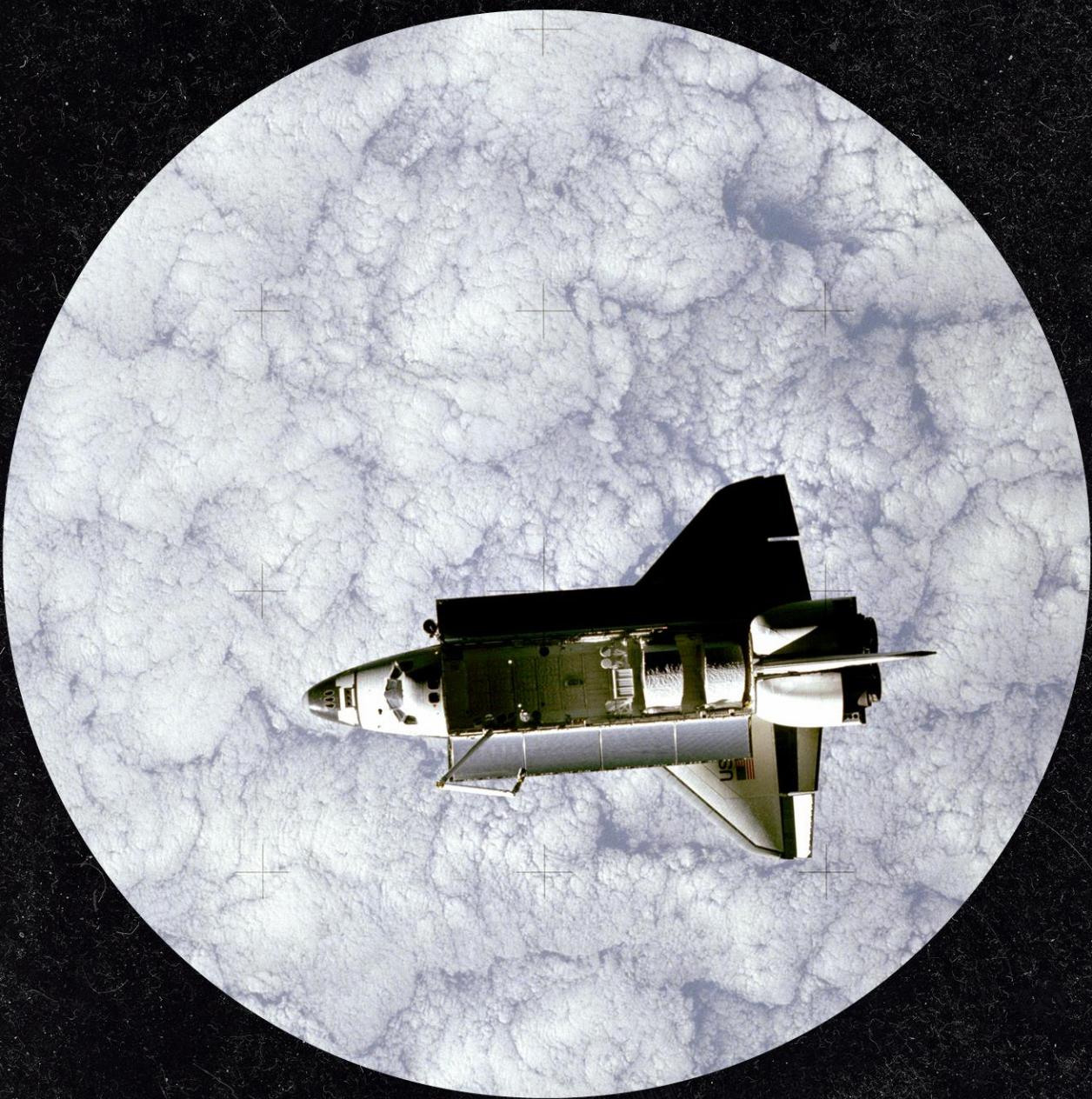
אני מכניס את שני המשתנים ל-`ReactDOM.render` לפי הסדר, ראשית JSX ו שנית המטרה שלי. זה הכל:

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

שמירת ה-HTML ופתחה שלו באמצעות הדף תציג לי `I am learning React!`

פרק 2

בנייה קומפוננטת פונקציה בסביבת העבודה הבסיסית



בנייה קומפוננטת פונקציית בסביבה

העובדת הבסיסית

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

בתרגיל בפרק הקודם יצרנו אפליקציית ריאקט שבה כתוב "I am learning React!".
אם אני רוצה לנתח כמה פעמים "I am learning React!" או, אני יכול לשכפל את ה-h2 ב-`h2` ב-`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
);
```

שים לב שאני צריך לעטוף את השכפולים שלי ב-`div` אחד מקיים, 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 שהוא זו שתדפיס עבורנו את:

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

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

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

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

```
<Greeting />
```

אלאמןט שסגור את עצמו. או כך:

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

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

ואיך הקוד המלא שלנו ייראה? כזה:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>React development environment</title>
  <script crossorigin
src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/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 הכיתוב בקומפוננטה או את האלמנט או להוסיף לקומפוננטה, אני יכול לעשות את זה במקום אחד בלבד, ובבת אחת השינוי הזה ישפייע על כל שימוש ושימוש בקומפוננטה בתוך האפליקציה שלי.

בתוך הקומפוננטה אני יכול להשתמש בעוד קומפוננטות, וזה חשוב מאד. הנה ניצור קומפוננטה נוספת שבה נשתמש 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 – היפשר באפליקציה או בתוך כל קומפוננטה אחרת. במקרה זהה מי משתמש בה הוא קומפוננטת Greeting. איך היא משתמשת בה? בדוק נמו 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 וapeuticו כמה וכמה פעמים, אבל פה הסתפקתי רק בפעם אחת.

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

תרגיל:

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

פתרונות:

```
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
);
```

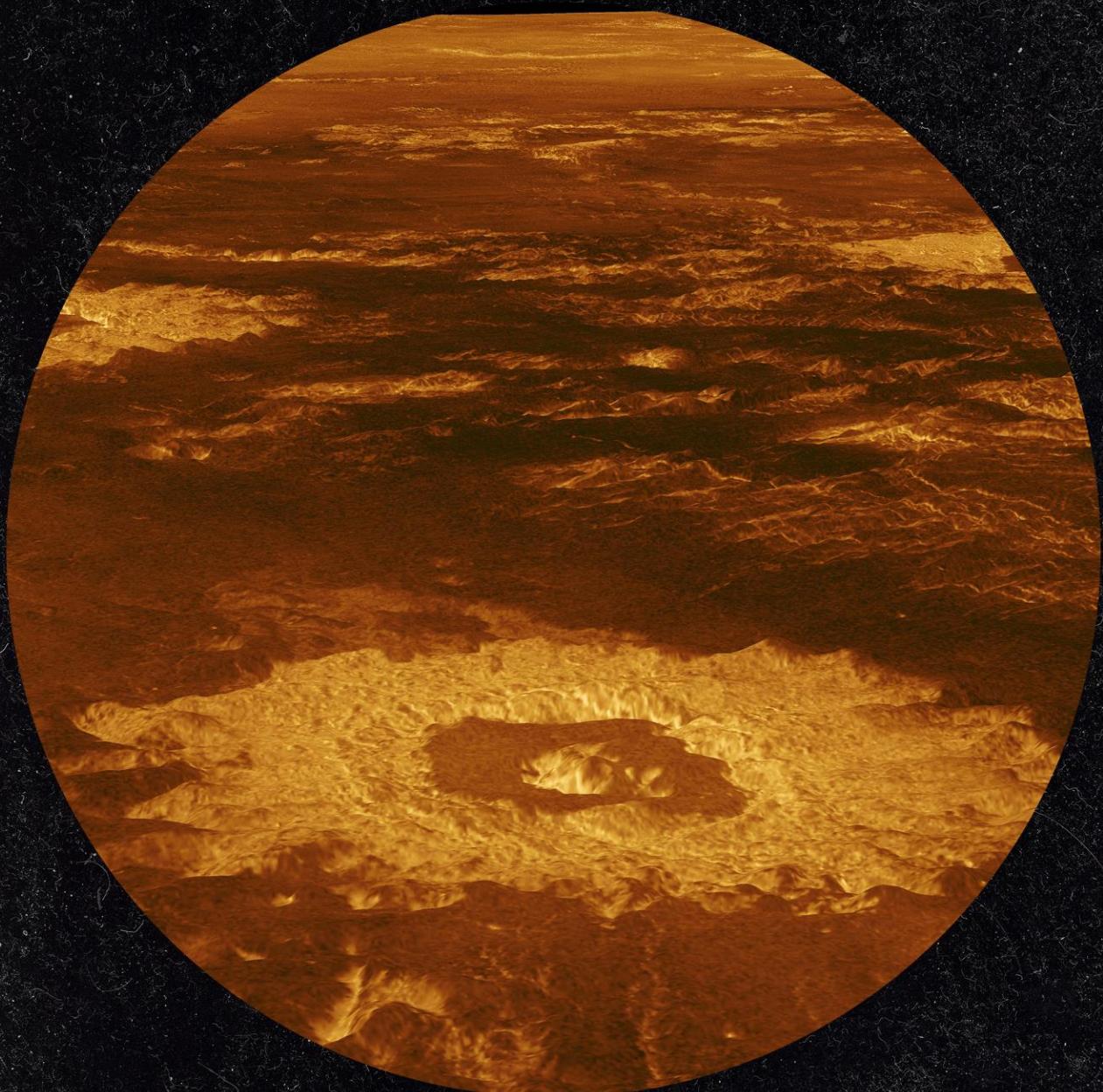
יצרתי שלוש קומפוננטות פשוטות. הראשונה והשנייה `Inigo`-`Greeting` מחזירות JSX פשוט.
 כדי לשים לב שב-JSX אני חייב לשים אלמנט אב אחד, במקרה זה בחרתי ב-`span`.

הקומפוננטות האלה הן קומפוננטות מסווג פונקצייה. כדאי לשום לב ששמן חייב להתחילה באות גדולה. מהרגע שהגדרתי אותן, אני יכול להשתמש בהן בכל קומפוננטה אחרת בדוק כמו כל אלמנט HTML אחר.

הקומפוננטה השלישית היא קומפוננטת Root והוא מכילה את שתי הקומפוננטות soigo ו-Greeting. גם פה, חשוב לציין, יש אלמנט אב אחד. גם הוא span. כל מה שנותר לי לעשות הוא להציב את הקומפוננטה השלישית, Root, באפליקציה שלי.

פרק 3

בניה של סביבת React אמיתית ומורכבת יותר באמצעות CREATE REACT APP בלבד



בנייה של סביבת ריאקט אמיתית ומורכבת יותר באמצעות Create React App

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

יחד עם ריאקט יש פרויקט שנקרו `Create React App`, המפותח ומתוחזק על ידי הצוות של ריאקט. הפרויקט הזה הוא `Bootstrapper`. כלומר הוא פרויקט של ריאקט עם המון תוספות שהצווות של ריאקט חשב שהן טובות וחינויות לפרויקט חדש. זו בעצם סביבת פיתוח מלאה הכוללת שירות שאפשר להפעיל בקלות מכל מחשב. סביבת הפיתוח הזו מבוססת על `Node.js`, אך לא נדרש ידע ממשוני ב-`Node.js` על מנת לעבוד בה.

אנו לא יכולים להישאר בעולם>Hello World, ולימוד בניית סביבת 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 חווה כל תקלה שתחוור בדרך (אם תחוור). חיפוש מהיר של התקלה בגוגל יסייע לכם מאוד.

התקנת `Node.js` על המחשב שלכם

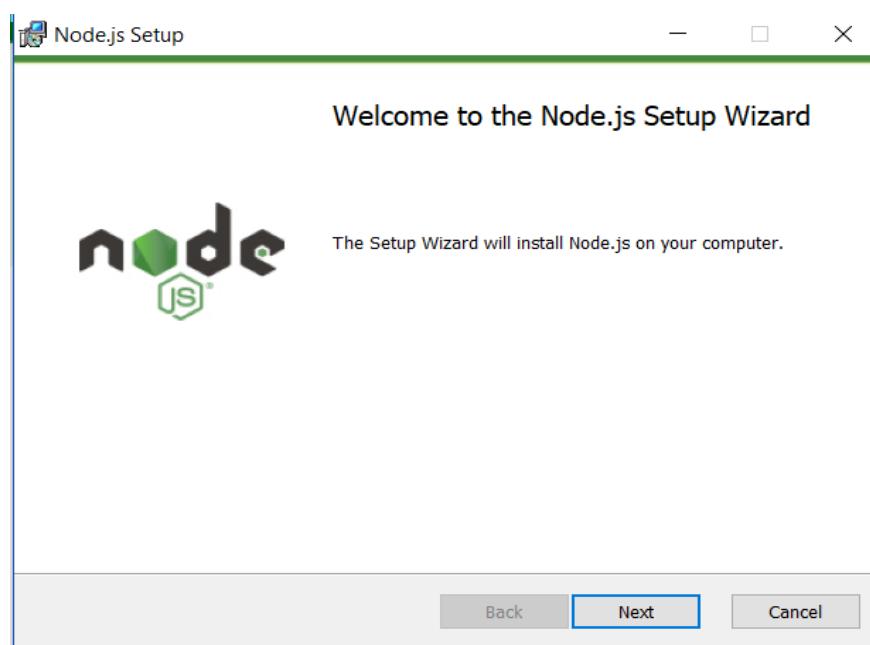
ראשית, אתם חייבים להתקין את `Node.js` על המחשב המקומי שלכם עובדים בו. `Node.js` הוא סביבה המאפשרת להריץ ג'אוوهסקריפט על גבי מערכת הפעלה, במקרה הזה מערכת הפעלה שלכם. ג'אוوهסקריפט, למי שלא יודע, היא שפה גמישה מאוד ואפשר להפעיל אותה לא רק

בסביבת הדפדפן, כמו שאנו עושים בפרויקט, אלא גם בסביבת מערכת הפעלה/ שרתים – ואת זה עושים באמצעות `Node.js`. הפלטפורמה זו ניתנת להתקנה בכל מערכת הפעלה שהיא ובקלות. בחרו את מערכת הפעלה שלכם והתקינו את `Node.js` לפי ההוראות.

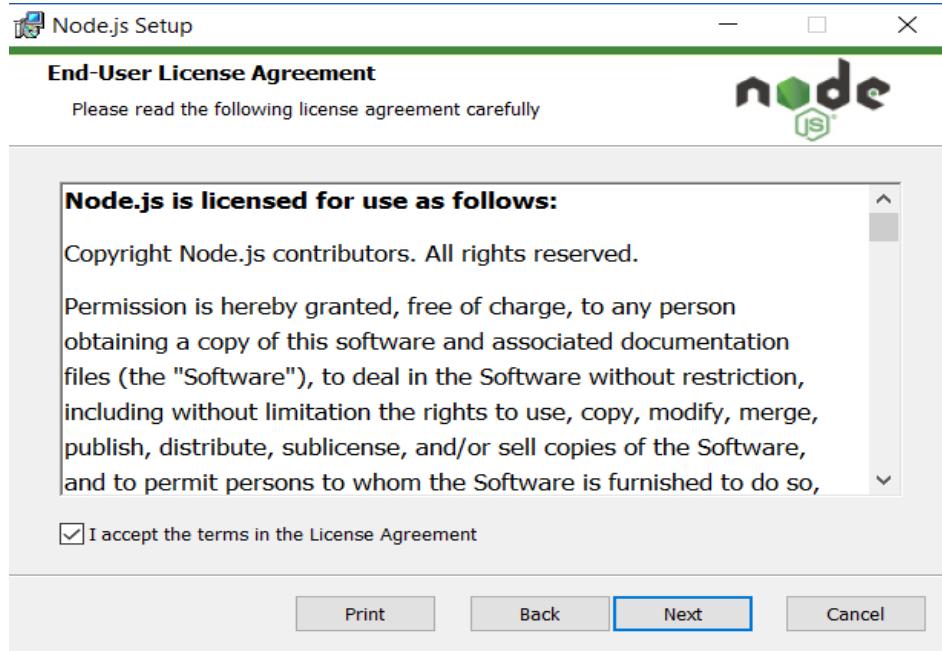
התקנה על חלונות

ההתקנה של `Node.js` על חלונות היא פשוטה. נקליך בגוגל `Node.js Download` או ניכנס אל: <https://nodejs.org/en/download/>

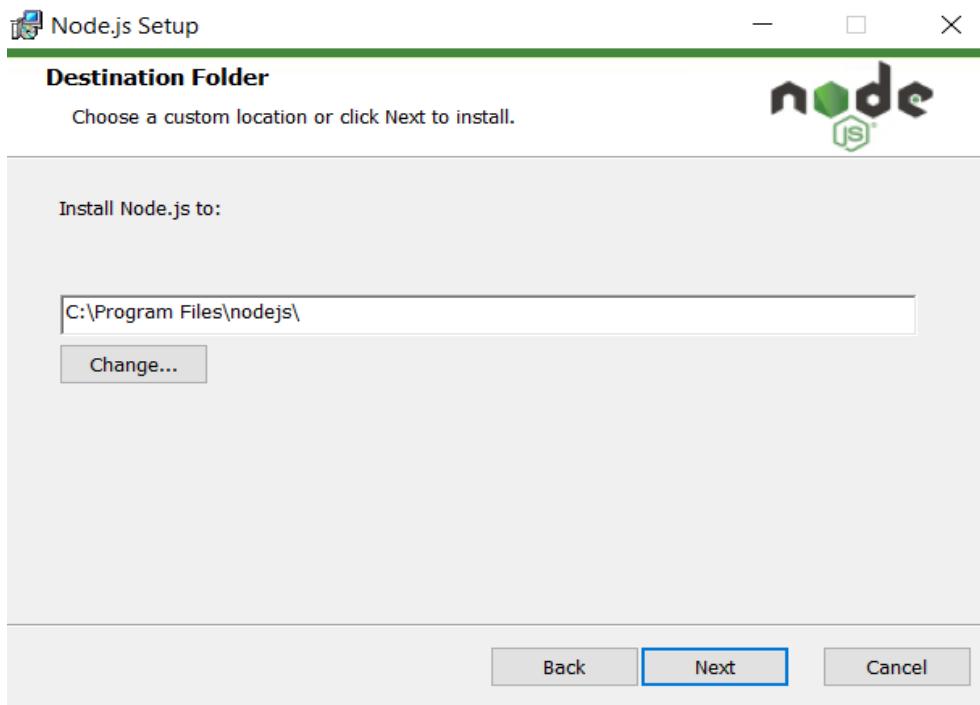
אנו נבחר בגרסת LTS – ראשית תיבות של "גרסה לטווח ארוך", ונבחר במערכת הפעלה שלנו – אם מדובר בחלונות, יש לנו `installer` נוח. מוריידים, לוחצים על התוכנה שיורדת:



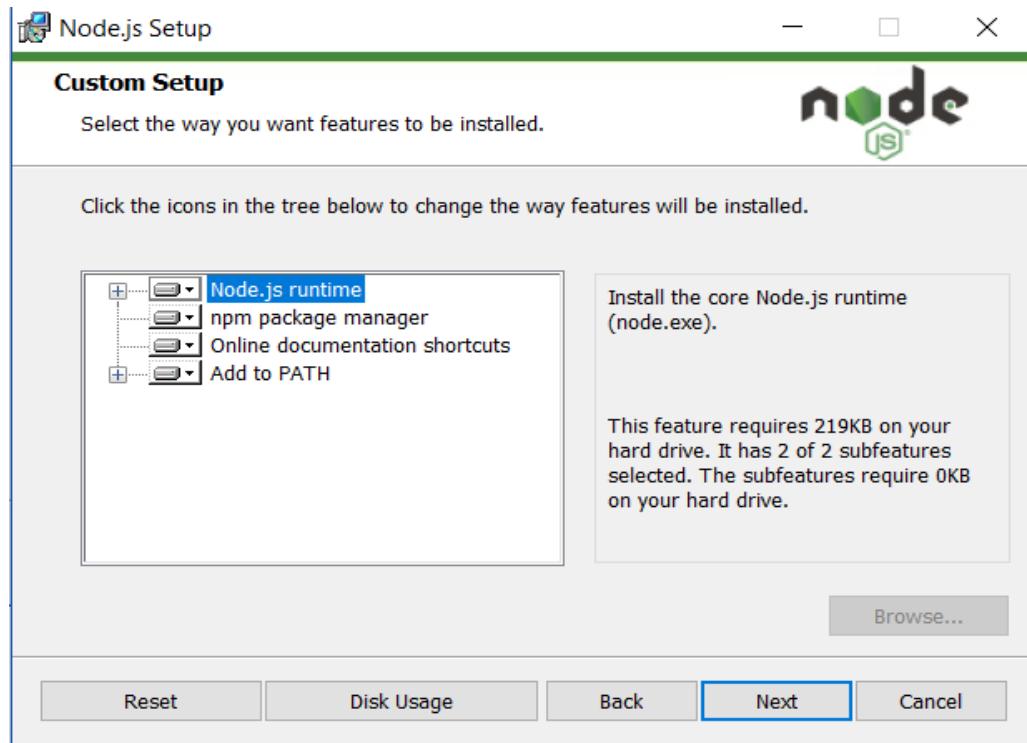
מקבלים את התנאים:



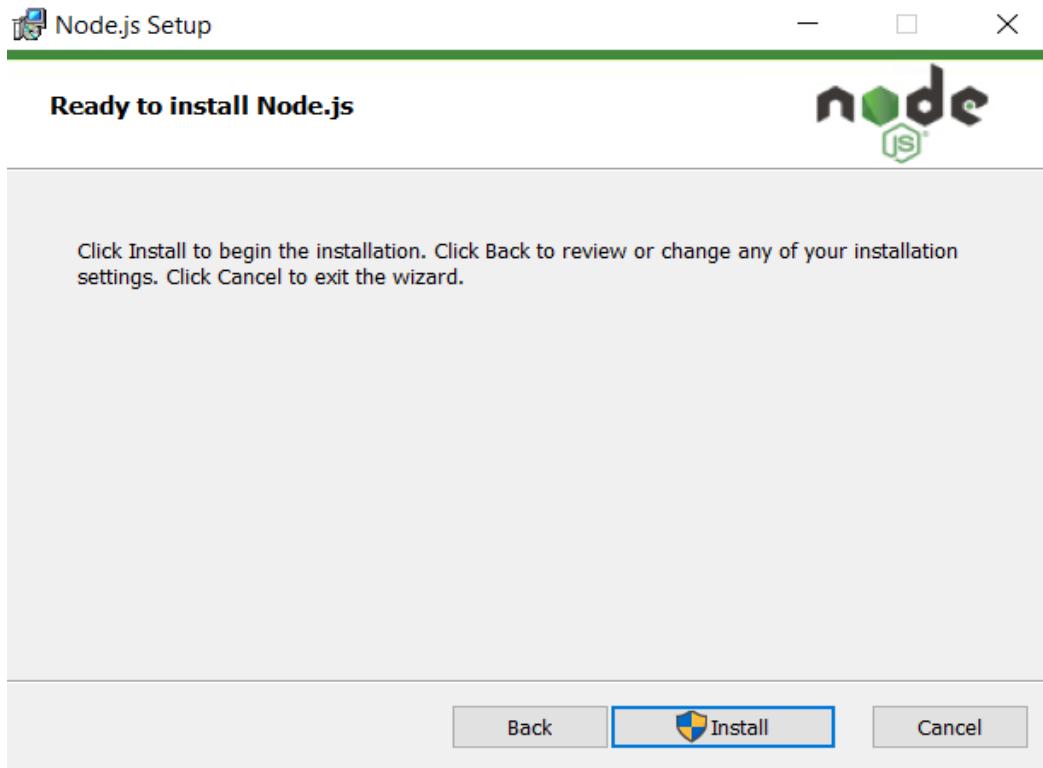
:Next לוחצים על



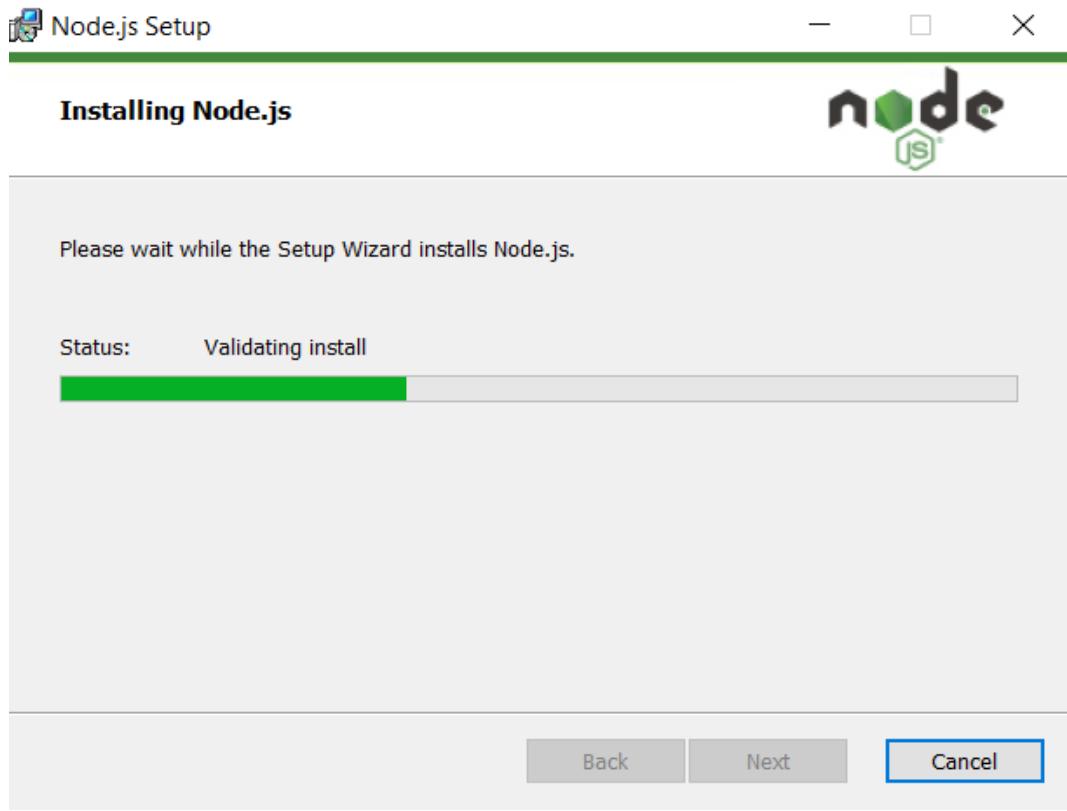
ושוב על :Next



לחיצה על Install לבסוף מתקין את התוכנה:



כל מה שנותר הוא לחכות לסיום ההתקנה:



התקנה על מק

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

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

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

התקנה על לינוקס

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

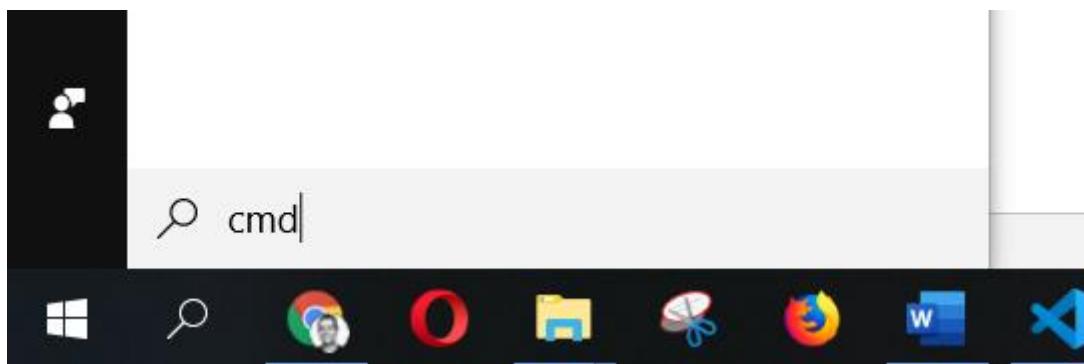
אני יוצא מנקודת הנחה שמשתמשים בלינוקס הם מיומנים בהרבה משתמשי חלונות ויודעים להתקין חבילת תוכנה ללא הסברים נוספים. נך או אחרת – לאחר ההתקנה, כניסה לטרמינל והקלדה של node יראו לכם את מספר הגרסה בדיקם כמו במק.

עבודה עם טרמינל

כמו כל אפליקציית js.Node, גם Create React App מצריכה אותנו לעבוד עם טרמינל. טרמינל הוא הממשק שבו אני מקlid פקודות למערכת הפעלה. כל מתכנת עובד עם טרמינל כאשר הוא מתחבר מרוחק לשרתים שונים וזה ידע שימושי למדוי. אנו נלמד בעת איך להתחבר לטרמינל ואין לעבוד איתנו בחלונות. אני לא מסביר על טרמינל במק או בלינוקס כי אני יוצא מנקודת הנחה שמי שיש לו את מערכות הפעלה האלה יודע איך להשתמש בסיסי בטרמינל.

הפעלת הטרמינל

בחולנות הגישה לטרמינל פשוטה למדוי. בחולנות 10 לוחצים על הזוכנית המגדלת, מקלידים cmd ווז לוחצים על אונטו.



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

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

C:\Users\barzik>
```

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

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

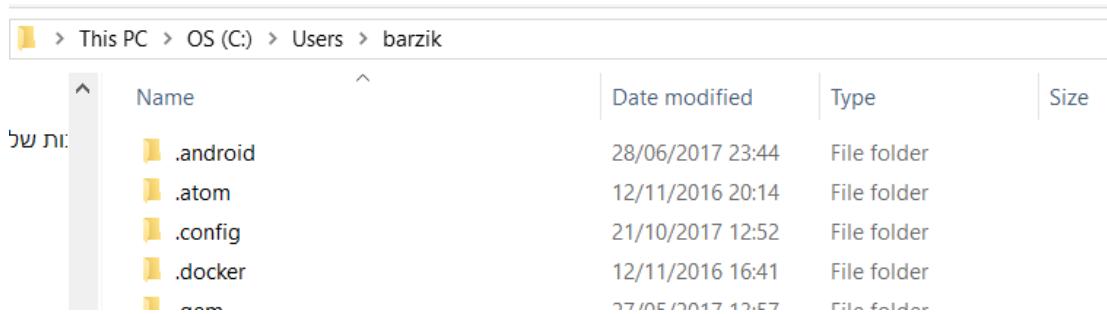
אצלכם המיקום יהיה שונה בהתאם לשם המשתמש שלכם. הבה ננסה להקליד פקודה. הקלידו dir והקישו על אנטר. אנטר בטרמינל הוא "שידור". אם תעשו את זה כמו שצرين, תראו מהו זהה:

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

Directory of C:\Users\barzik

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

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



Name	Date modified	Type	Size
זהות של			
.android	28/06/2017 23:44	File folder	
.atom	12/11/2016 20:14	File folder	
.config	21/10/2017 12:52	File folder	
.docker	12/11/2016 16:41	File folder	
.vscode	7/6/2017 23:00 23/06/2017 23:00	File folder	

ניוט בטרמינל

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

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

אפשר "לעלות" לתיקייה אחת למעלה באמצעות .. cd.

נסו לעלות שוב ושוב עד שתגינו לתיקייה הראשית, הלווא היא: c.

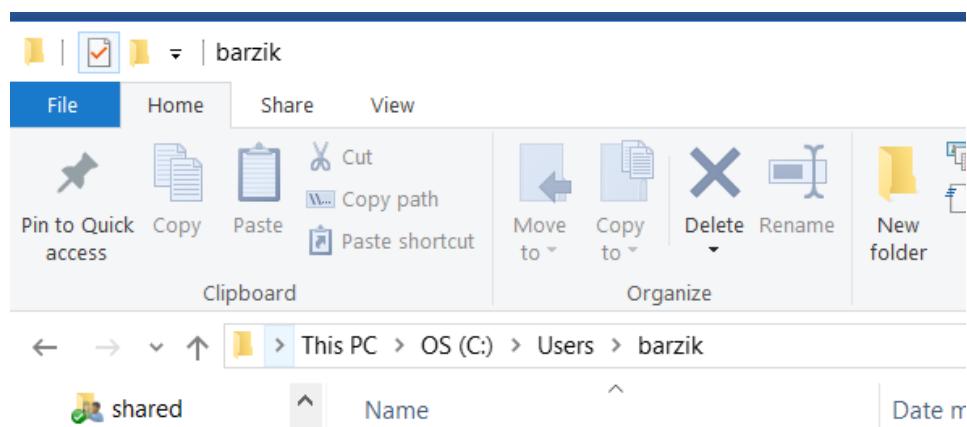
אנו יכולים לנוט שוב בחזרה באמצעות cd ושם התיקייה. אפשר גם להקליד ישירות את הנתיב:

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

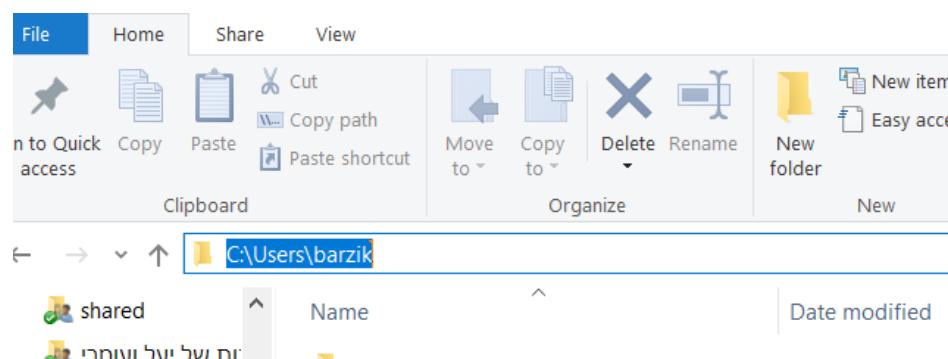
טיפ קטן – אפשר להקליד tab כדי לבצע השלמה.

מציאת מקומות בטרמינל דרך חלונות

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



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



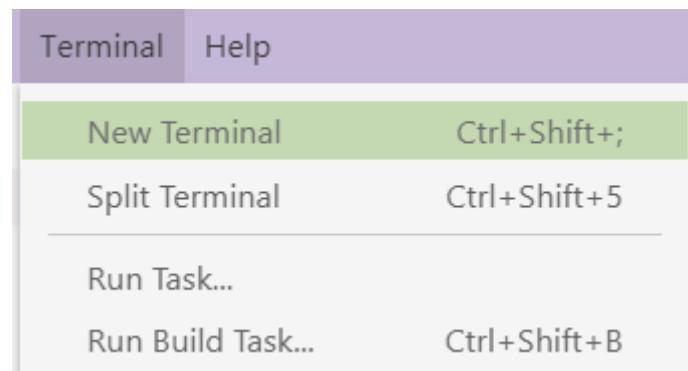
אני יכול להעתיק את הנתיב אל הטרמינל. להקליד:
`cd C:\Users\barzik`

ואז ללחוץ על אנטר ולהגיע אל היעד המבוקש.

טרמינל ב-Visual Studio Code

אפשר לעבוד עם הטרמינל דרך ה-IDE החינמי Visual Studio Code, שומומלץ לכל מפתח ג'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 ובחורו ב-`New Terminal`:



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

בדיקות גרסת `js.Node` דרך הטרמינל

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

```
node -v
```

אם קיבל שגיאה, סימן שההתקנה לא הייתה תקינה. נסו להפעיל מחדש את המחשב או להתקין מחדש את `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 מקלידים `-v` בטרמינל אתם מקבלים את הגרסה, אפשר להתקדם לשלב הבא – עבודה עם Create React App.

עבודה עם Create React App

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

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

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

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

```
npx create-react-app my-app
```

אך היא פקודה הפעלה של Node.js.

`create-react-app` הוא שם התוכנה שאנחנו מפעילים, `my-app` הוא שם האפליקציה שלנו. כנסנו לתוכנה אמיתית אנו נקרא לה מן הסתם בשם `my-app` – האפליקציה שלי.

אם יש לכם Node.js במערכת והכל תקין, ההתקנה תעבור בקלות. היא נמשכת כמה דקות טבות, אין מה לחושש. במהלך הדקות האלו התוכנה מורידה את כל המרכיבים מהרשת 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.
נוצרה לכם תיקיה שנקראת app-my. היכנסו אליה באמצעות:

`cd my-app`

והפעילו את המערכת החדשה שלכם באמצעות:

`npm start`

מדובר בפקודה של `Node.js` שפשוט מפעילה סקורייפט שנקרא `start`. אם הכל תקין, הדבר הראשון שתשים לב אליו הוא שנפתח לכם של דףון שמניל את האפליקציה שלכם! זה לא קסם. מה שמתהיל `Node.js` עווה הוא ליצור שירות על המחשב שלכם מאשר הקלעים, לטען את רכיבי הריאקט ו אז לפתח דףון שמכoon אל השרת המקומי שלכם יש מuin. אם

תסתכלו על כתובות הדף הנראות שמדובר בכתובת של localhost:3000. הכתובת הזו מחולקת לשני חלקים. localhost 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 סביבת לימוד ופיתוח מצוינת.

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

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

אפליקציית הריאקט גם היא כבר מוגדרת ב-`index.js` וכן כוללת 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 מתבצעת ההפניה? באמצעות import, שנרכיב 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';

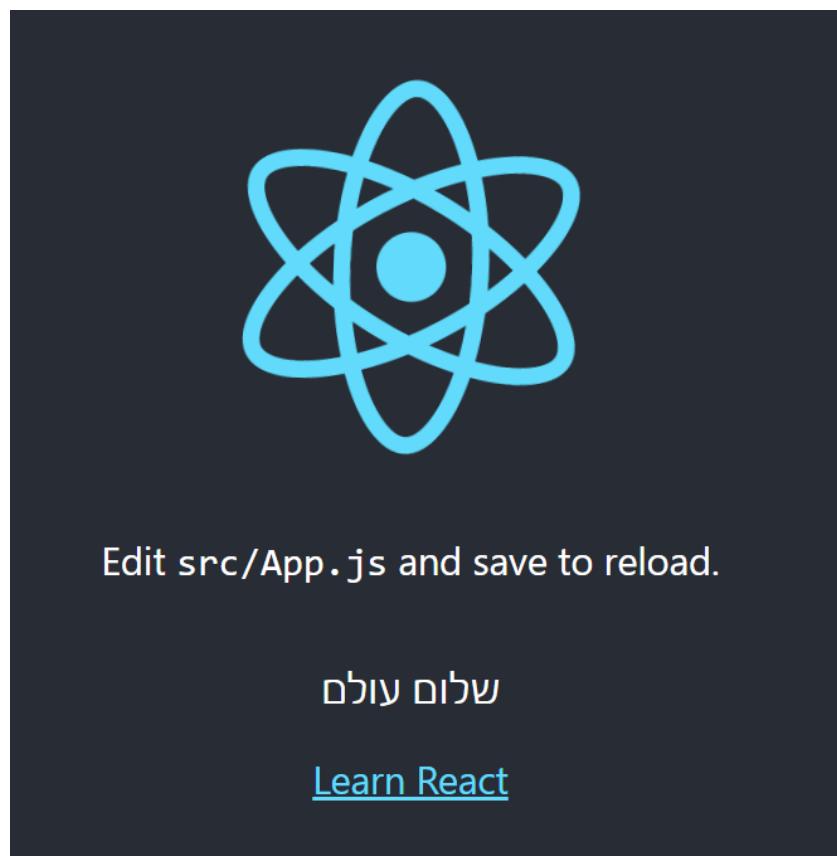
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"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

export default App;
```

אם נשנה משהו בקומפוננטה, למשל נוסיף שורה כזו:

```
<p>
שלום עולם
</p>
```

ונשмарו, נוכל לראות שבמיטה קסם, גם האפליקציה שלנו השתנתה ומודפיע בה "שלום עולם".



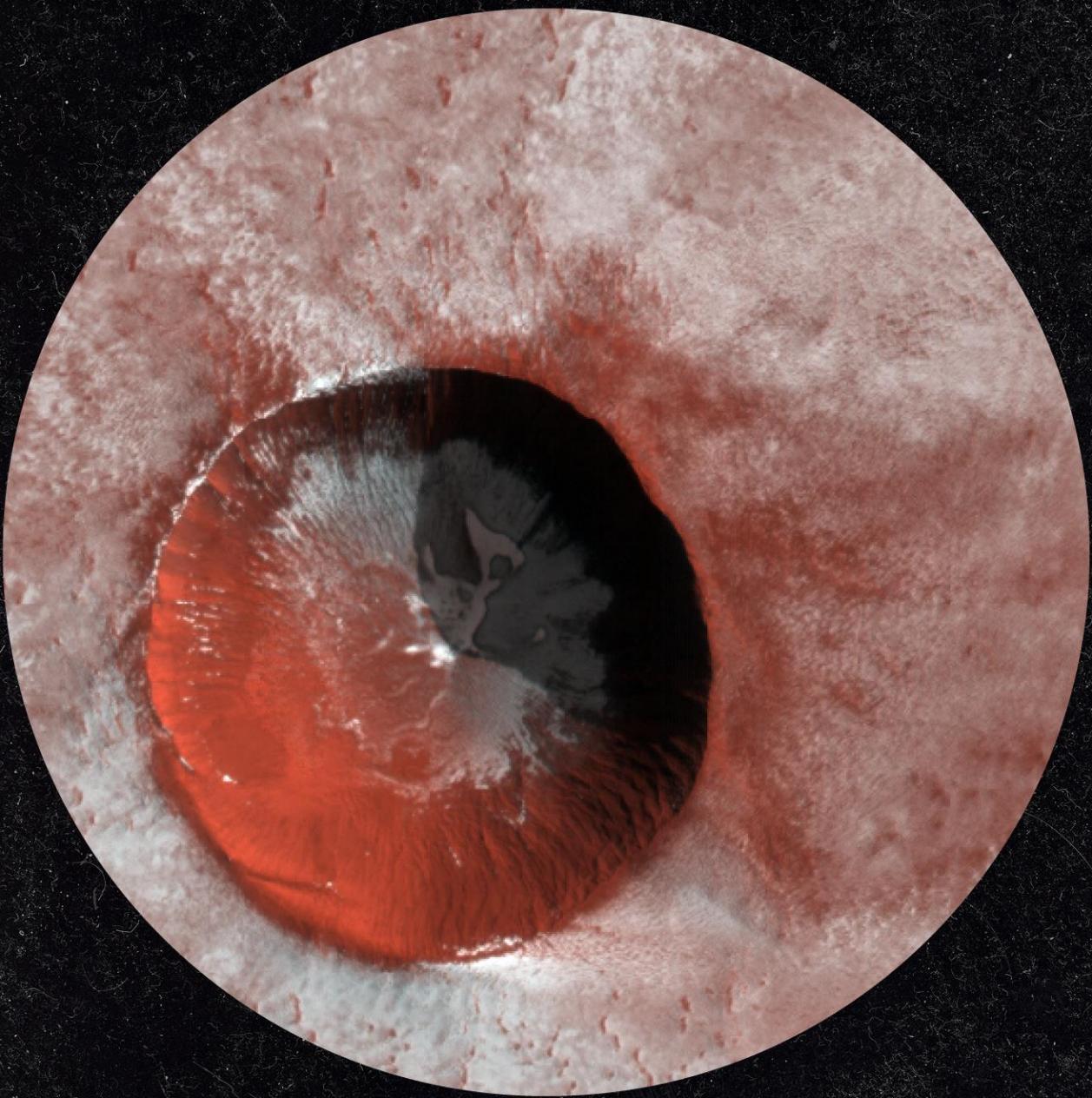
זה קורה כי השרת מażין לכל הקבצים בתיקייה וגם אחד מהם משתנה הוא טוען את עצמו מחדש. נשמע מסובך ופלאי, אם כי זה לא מסובך להבנה וכל מתכנת שמכיר `js` `Node` יוכל לייצר זהה דבר. אבל אנחנו לא צריכים להתאים – `Create React App` יכולה לעשות את זה עבורנו. ממש נפלא. בסופה של התהליך יש לנו אפליקציה בסיסית שעובדת ונחנו יכולים ליצור לה קומפוננטה ראשונה.

קשיים ותקלות

אם נתקלتم בבעיות בתהיליך ההתקנה של Create React App, אל דאגה! יש לפעמים בעיות שנובעות משינויים במערכת הפעלה, אנטי-וירוס שפותאים מתעורר או בעיה אחרת שנובעת מכך שמערכות הפעלה שלנו שונות. אבל כדאי מאוד לא לוותר. אם יש הודעה שגיאה – פשוט חפשו אותה בגוגל. Create React App היא כל כך פופולרית, עד שיש תיעוד רב מאוד לגבייה בראשת גם תיעוד של כל תקלה אחרת. קיבלתם שגיאה אדומה ומפחדה? חפשו אותה בגוגל, התיעיצו לגבייה בקובץ פיסבוק או באתר StackOverflow. אל תלגנו על בניית סביבת העבודה זו כי היא הבסיס העיקרי לעובדה עם ריאקט.

פרק 4

כתיבת קומפוננטה ו Asheona CREATE REACT APP-1



כתבת 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 אמיתיות יותר. אנחנו לא נכניס את הקומפוננטה שלנו באותוקובץ של `App.js`. כלל אצבע חשוב מאוד: כל קומפוננטה נמצאת בקובץ נפרד משלה ועומדת בראשות עצמה. מקובל מאוד שם הקובץ יהיה זהה לשם הקומפוננטה ויתחילה באות גדולה. במערכות גדולות מקובל להציג כל קומפוננטה בתיקייה משלה.

הבה ניצור קומפוננטה בשם `Greeting` שבה יהיה כתוב "שלום עולם". ראשית – הקובץ: ניצור `Greeting.js` תחת תיקיית `src`. שימו לב שם הקובץ מתייחל ב-`G` גדולה. הקומפוננטה זו אינה שונה כלל מקומפוננטה שלמדנו לעבוד איתה 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` ומגיע לפני הפונקציה שמיצאים. אנו נהchetib על `import\export` בהמשך, אבל גם בלי להבין לעומק זה הגיוני – אנחנו שמים את הקומפוננטה בקובץ נפרד – ולקובץ זהה אנו עושים `import`. אנחנו חייבים לעשות `export` בצד השני. את `export` עושים כך:

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

אבל אם נשים רק את זה, נראה שהוא מקבלים שגיאה:

`'React' must be in scope when using JSX react/react-in-jsx-scope`

השגיאה זו מرمצת על כך שימושו חסר. מה חסר? במקרה זהה הריאקט עצמו. בעוד בסביבת הלימוד הפשטota שעליה למדנו, ריאקט נמצא בכל מקום כי הכל היה בקובץ אחד, וכך אנחנו חייבים לכלול בכל קומפוננטה וקומפוננטה את כל הרכיבים שאנו צריכים להפעלה. במקרה זהה: ריאקט. אנו חייבים "לייבא" את ריאקט ועושים את זה באמצעות הבא:

```
import React from 'react';

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

از יש לנו כאן ייבוא של ריאקט, שהוא בעצם פשוט וחינוי בכל קומפוננטה, יש לנו ייצוא, שייהי חשוב כשרצה להשתמש בקומפוננטה, ויש לנו את הקומפוננטה שלנו, שבמקרה זהה לא עשו המון אלא רק מחזירה טקסט. והכל נמצא בקובץ בודד אחד שנקרא `Greeting.js`.

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

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 />
        <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 בתוך קובץ ושאנו חנו משתמשים ב-`import\export` מסתורי לא אמרור להפריע לנו. בהמשך נלמד עוד על `import\export`.

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



הקפיצה מסביבה לימודית פשוטה לסביבת פיתוח מורכבת נראה קשה וمضיכה, אבל ככל שתתאמנו יותר – כך תתרגלו אליה. יש בה המון דברים טובים – ראשית יש לנו את babel שmagical, eslint, ביחסם, יש לנו שירות פיתוח, בדיקות (בהמשך נלמד על הבדיקות) וגם בודק תקינות קוד של `slint`, שמתՐיע על בעיות שימושינו מהאפליקציה שלנו לרוֹץ. והכל באמת בא מהקופסה. אז גם אם זה נראה מורכב, נא לא להיבהל ולתרgal על הסביבה הזאת. זו הסביבה שבסוףו של יום תפגשו במקומות העבודה.

תרגיל:

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

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

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

פתרונות:

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

Welcome.js

Inigo.js

Greeting.js

הkomפוננטות Greeting.js ו-Inigo.js הן קומפוננטות באמת פשוטות. הדבר היחיד שאנו צריכים לזכור זה את import React from 'react' export default () {...}. כך הן נראות: Inigo.js

```
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 מתוכן Welcome.js ומשתמש בהן ב- JSX כרגע. בדוק כדי ששהשתמשתי בסביבת הפיתוח הפשוטה. כל מה שאנו צריך לזכור הוא לבצע import. זה הכל. כך sz תיראה:

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

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

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

השוני בין סביבת העבודה הפשוטה יותר זו של `Create React App` הוא ה-`import\export`. בעצם כל קומפוננטה עשו `import` לדברים שהוא צריכה ו-`export` לעצמה, כדי שאחרות יוכל להשתמש בה.

פרק 5

EXPORT / IMPORT



Import\Export

מדובר בחלק אינהרנטי מג'אווהסקרייפט, שחשיבותו לכל סקריפט בג'אווהסקרייפט מורכבת. המנגנון זהה מאפשר לנו לעבוד עם כמה וכמה קבועים ולנהל את הקוד שלנו بكلות. בעבר היה מקובל לכתוב את כל הקוד בקובץ ארוך מאוד וזה כמובן קשה מאוד לניהול ולביצועה. כפי שראינו בסביבת הפיתוח המודרנית, כדאי לעבוד עם חלוקה של קבועים עם `import` ו-`export`.

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

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

הבה נדגים ייזוא פשוט. ניצור קובץ בשם `Vars.js` שיכיל משתנים. ניצור בפרויקט של Create React App קובץ צזה ושם נגדיר משתנה וניצא אותו. בתיקיית `src` ניצור קובץ `Vars.js` וכל מה שיהיה בו זה:

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

שימוש לב שאין פה קלאס, אין פה משתנים – אני מיצא טקסט בלבד. כדי להשתמש בו, אני רק צריך לעשות `import`. ניכנס ל-`App.js`, ניבא אותו ונדייס אותו בקונסולה. הקוד שלנו ייראה 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 לשכוח שעם כל הכבוד לקומפוננטות, הן כתובות בג'אוועסקריפט ומנתנегות כמו כל קוד אחר של ג'אוועסקריפט. אם אני אשים `console.log` בפונקציה של הקומפוננטה אני אוכל לראות את המשתנה בקונסולה של הדפדפן. אם אני אפתח את כל הפתחים, אני אכן אראה את הטקסט שעשיתי לו `export`:

```

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

JS App.js      ...
src > JS 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` שעלייה נלמד בהמשך.

אנחנו יכולים לעשות `export` לפונקציה. למשל:

```

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

export default foo;

```

כדי לשימוש לב שמדובר בפונקציה רגילה ולא בפונקציית קומפוננטה (רואים את זה כי הוא לא מתחילה באות גדולה ולא מחזירה JSX). כשאנו מיבא אותה (שוב, באיזה שם שאני רוצה) אני צריך להפעיל אותה בדרך כללשי. בדוגמה זו אני מבצע ייבוא בשורה הרכיבית 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 בדיקת הדריך ליבא וליצא פונקציה שהיא קומפוננטה בריאקט. קומפוננטה בריאקט היא בסופו של דבר פונקציה ג'אוועסקרופטית לכל דבר, רק צזו משתמשת ב-XJS. אנחנו מיבאים אותה ומשתמשים בה ב-XJS (בניגוד לפונקציה רגילה, שלא אנו קוראים). זה הכל.

ייבוא מנתיבים אחרים

עד עכשיו היינו קוד שנמצא בבדיקה באוטה תקינה של הקומפוננטה שבה השתמשנו בקוד המיו בא. אבל מה קורה אם הקוד שלנו נמצא נמצוא בתיקייה אחרת? במקרה זה אנו צריכים להקליד את הנתיב היחסי של התיקייה שלנו ביחס לקובץ שבו אנו נמצאים.

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

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

מהה כתובות זו מורכבת?

החלק הראשון - ./ - מסמן את המיקום של התיקייה שלנו, הקובץ שבו אני כותב את הקוד. מהנקודה הזו הכל מחייב לנו צורך צריים לכתוב נתיב ייחודי מהמקום שבו אנו נמצאים.

החלק השני – תת-תיקייה components.

החלק השלישי – שם הקובץ המקורי. אם מדובר בסיוםת של `js` הוא לא הכרחי:

```

1 import React from 'react';
2 import logo from './logo.svg';
3 import './App.css';
4 import myVar from './components/Vars.js';
5
6 function App() {
7   console.log(myVar());
8   return (
9     <div className="App">

```

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

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

הענין מסתבכים כאשר התיקייה שאנחנו רוצים לייבא ממנה היא לא תת-תיקייה של התיקייה הנוכחית שאנחנו נמצאים בה, אלא תיקייה שנמצאת מעל או תקיה אחת. במקרה הזה אנו נעזרים

ב-./. שואמר "לעלות תיקיה אחות למעלה".

למשל, הנה נניח שיש לאפליקציה שלנו מבנה תיקיות כזה:

```

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

```

אני רוצה ש-`js`-`App` תייבא מ-`js`-`MyVar`. איך אני עושה את זה? אני צריך "עלות" תקינה אחת למעלה מ-`components` אל `src` ואז לרדת אל תקינת `components`. את המסע הזה אני מתאר באמצעות `import` כ-:

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

הבה נראה את המסע:
 חלק א' – /. אנו מתחילה מהמקום שלנו.
 חלק ב' – ../ עולמים תקינה אחת ל-`src`.
 חלק ג' – יורדים אל תקינת `components`.
 חלק ד' – קובץ `js`-`MyVar`.

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

אין חובה להשתמש בסימנת הקובץ `js`

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

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

יצוא של כמה משתנים

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

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

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

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

או:

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

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

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

והיבוא יijk 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'אווהסקריפט, בפועל מי שדואג לטיענה של המשאבים

ב-App Create React שעה אנו לומדים וכן בחלק מאפליקציות הוב המודרניות הוא הספרייה וויבפָאַק (webpack). הספרייה זו היא מודול מבוסס Node.js ואחרראית על תהליך הבילד (build) של האפליקציה, תהליך שגם במסגרת מתנהלת טעינת הקבצים.

אנו מסוגלים להשתמש ב-`import` גם כדי לייבא תמונות וקובצי CSS. אפשר לראות ב-`App.js`:

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

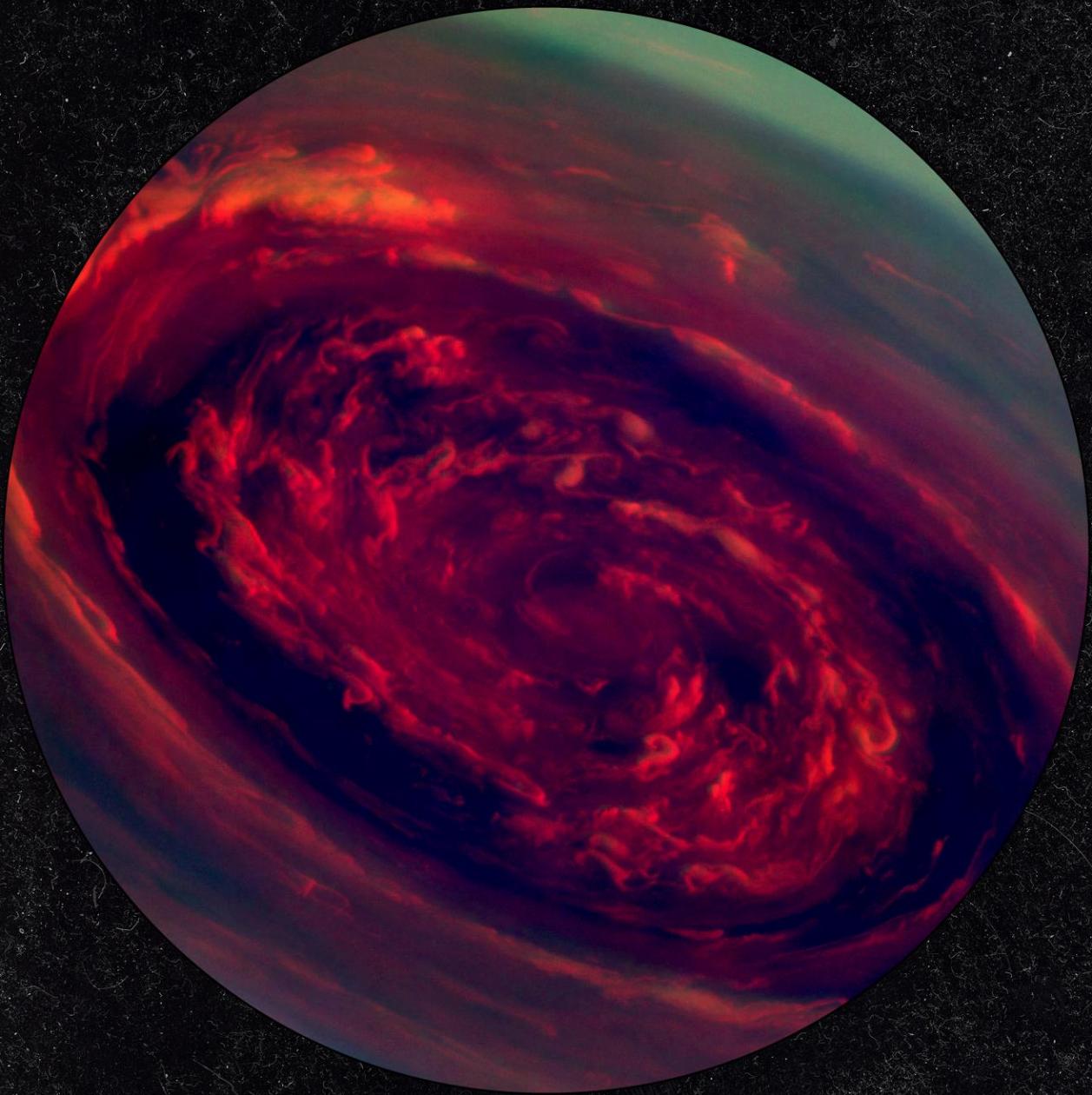
function App() {
  const someVar = true;
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
      </header>
    </div>
  );
}

export default App;
```

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

פרק 6

JSX



JSX

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

הכלל הוא פשוט – כל ביטוי שמופיע בין סוגרים מסוללים { } ונמצא ב-**JSX** בעצם נחשב לג'אווהסקריפט ורץ כג'אווהסקריפט והותוצאה מופיעה בקומפוננטה. ככלומר, אם יש לי משתנים ואני רוצה להדפיס 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 מכיריז על משתנה `myVar` וublisher אותו ל-**JSX**. הקומפוננטה כבר תציג להציג אותו. אם אציג את הקומפוננטה באמצעות מבט בדף (בכתובת `localhost:3000`) אני אראה את הטקסט של המשתנה.

מובן שאני יכול להציג סוגריים מסווגלים כמו פעים 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 זה 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, אני גם יכול לשים ב-JSX שנקנס לתוך משתנים המשתנים בתוך סוגרים מסולסלים.
העניין הוא ש-JSX הוא ג'אויסקcript טהור ויש להתייחס אליו בהתאם. אפשר לעשות אותו
הכל בדיק נמו בג'אויסקcript. למשל, להכניס בתוכו ביטוי:

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

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

אבל הכוח של JSX הוא לא בפעולות נלוימות 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 מסולסים בתוך ה-`return` כדי שנוכל לעבוד עם ביטויי ג'אויסקייפט.

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

בתוך הסוגרים האלה אנו יכולים לכתוב כל ביטוי ג'אויסקייפטי 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 חצ' פטוטה שאנו מכירום ומחזירה גם היא JSX. על מנת שגם שם הילד יודפס בין ה-chads, אנו מקופים גם אותו בסוגרים מסולסלים, 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 ידפיס לי אותם באופן אוטומטי.

```
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 ייצורתי מערך של אלמנטים. כדי להדפיס אותו אני לא חייב להשתמש ב-map אלא רק לשימושו ב- JSX והוא יודפס אוטומטית. זה לא יעבוד עם אובייקט, רק עם מערך.

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

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

```
function App() {
  return (
    
  );
}
```

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

```
function App() {
  return (
    
  );
}
```

ההבדל נראה קטן, אך הוא ממשוני. לא מדוברפה ב-HTML אלא ב-XSL, ג'אווהסקריפט שעבוד עם אובייקטי XML, שאליהם גם האובייקטים ש-HTML עובדים איתם.

רשימות ב-XSL

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

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

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

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

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

אם הקוד הזה:

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

נראה לכם לא מובן, כדאי ללמידה שוב על פונקציית map ועל פונקציית `cz`, משומם שהן קריטיות להבנה ומשתמשים בהן המון בריект.

תרגיל:

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

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

פתרון:

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

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

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

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

תזכורת:

הדף לולאת `for` מ-10 עד 0 עובדת כך:

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

פתרון:

```
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 `<a>` עם ? . כדאי לשים לב שכיוון שאנו רוצים שה-`key` יהיה את ערכו – אנו שמים אותו בסוגרים המסוללים. לא נשכח גם לשים `index` ב-`key`.

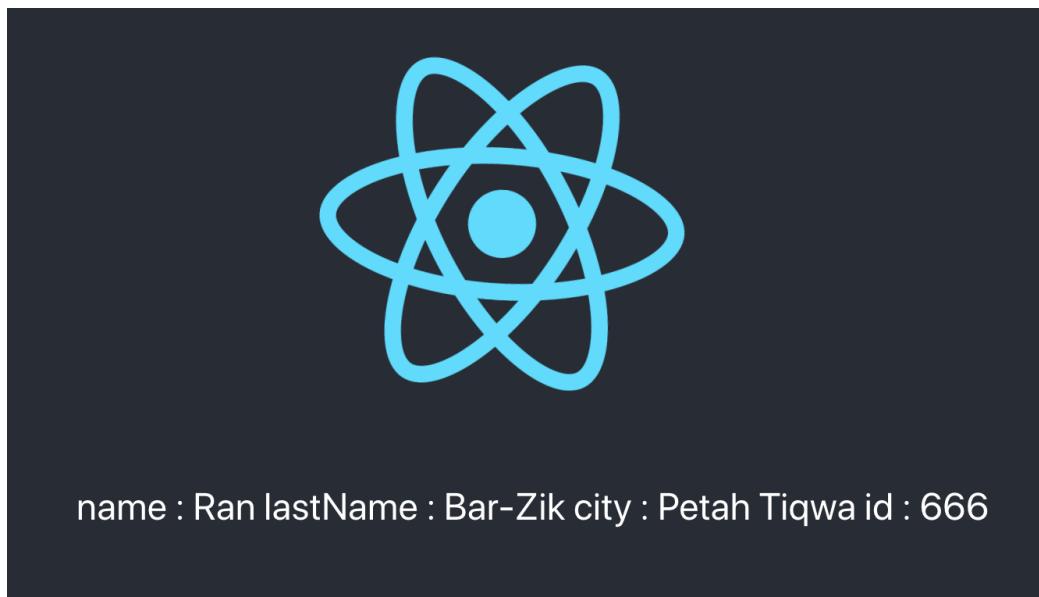
כל מה שנותר לנו לעשות הוא לחת את המערך ולהדפיס אותו. כאמור, JSX יודע לחת מערך ולטפל בו אוטומטית.

תרגיל:

נתון אובייקט משתמש:

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

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



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

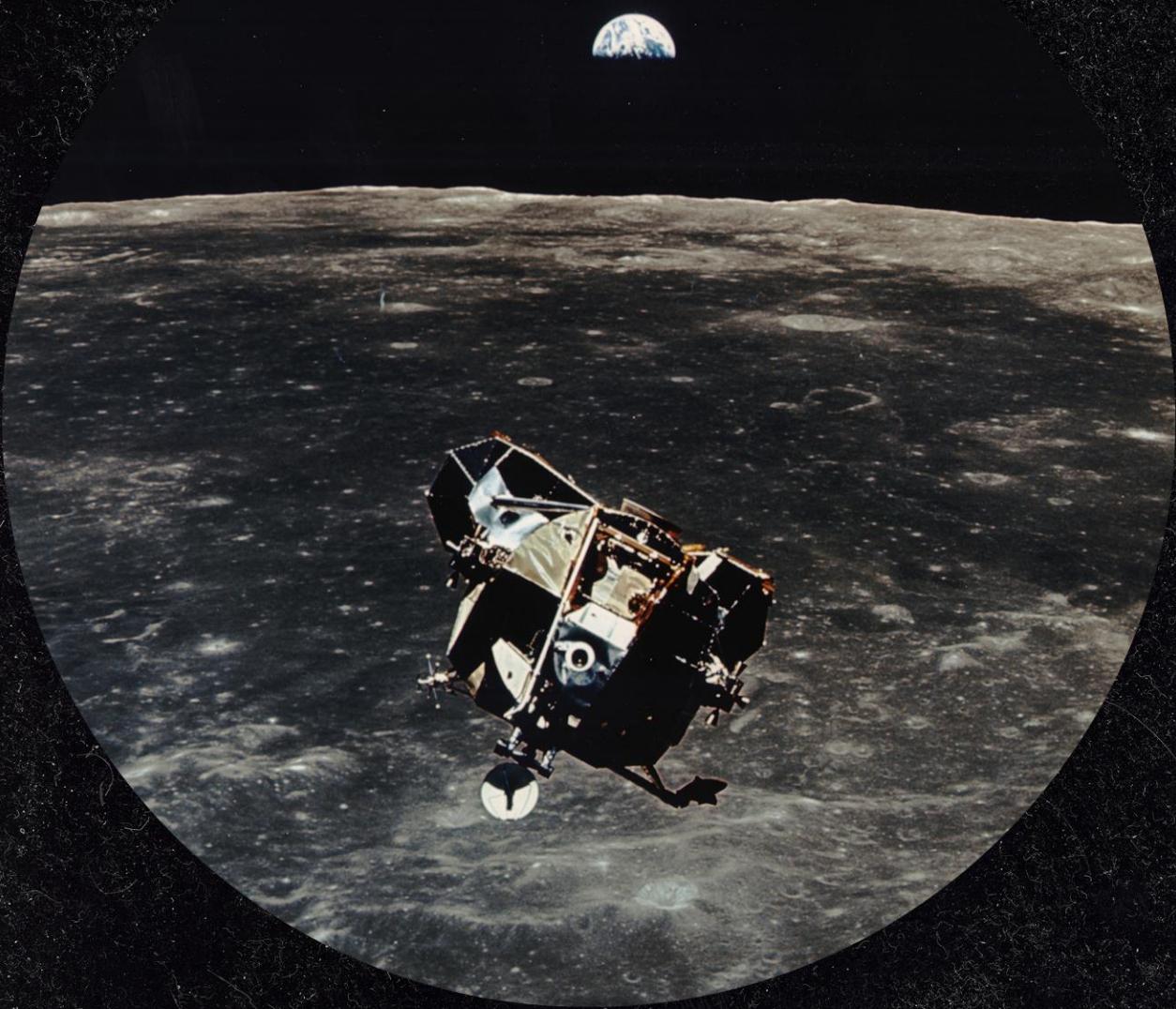
function App() {
  const user = {
    name: 'Ran',
    lastName: 'Bar-Zik',
    city: 'Petah Tiqwa',
    id: '666',
  }
  const userArray = [];
  Object.keys(user).map((value ,index) => {
    userArray.push(<span key={index}>{value} : {user[value]}</span>)
  })
  return (
    <div className="App">
      <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 ה-`XJS` והכנסתי אותו לתוך מערך באמצעות `push`. השלב הבא הוא פשוט לשום את המערך זהה בתוך ה-`XJS` ולסמן עליו שירנדר הכל.

פרק 7

הומפוננטה עם תוכנות (PROPS)



קומפוננטה עם תכונות (props)

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

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

ראשית, ניצור את הקומפוננטה הבסיסית בקובץ `Welcome.jsx`. אני רוצה שתשימו לב שכעת, כשהאנחנו עובדים עם קומפוננטות מסודרות, נהוג לנתח את סימנת הקומפוננטות כ-`-xsj` ולא `-csj`. 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 צריכה לצרוך את הקומפוננטה זו, אני אצורך אותה עם `import` (שים לב שאלה אctrar לשלוט את הנתיב, כפי שלמדנו בפרק על `import` ו-`export`):

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

ואשתמש בה בכל מקום פה:

```
<Welcome />
```

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

```
function Welcome(props)
```

ה-`props` זהה הוא בעצם אובייקט הקלט שלנו. יש בו כל הפרמטרים שנעביר לקומפוננטה. איך אנו מעבירים? באמצעות התכונה של הקומפוננטה, שנראית בדיקן כמו תכונה של HTML. הנה, כך:

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

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

```
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 תגי HTML נקראות באנגלית `HTML attributes`. למי שלא מכיר כל כך לHTML, תכונות הן בעצם הדרך שלנו להעביר מידע לתגית HTML כמו קישור. למשל:

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

ה-`title` וה-`href` נקראים "תכונות", וזה הדרך שלנו לתקשר עם רכיבי HTML ולשנות אותם. אותו הדבר עם קומponentות 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 זהה 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 `-props` הוא לא במשЛОח מחרוזות טקסט. אני יכול לשלווח הכל. ב-`JSX` אני יכול לשלווח למשל אובייקטים שלמים. בואו נניח שיש לי אובייקט משתמש עם שם ותואר, שהוא:

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

איך אני שולח את האובייקט הזה?

אני יכול לעשות משהו זהה:

```
<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 בכך מיוחדת שבה אנו לוקחים אובייקט ומפרקים אותו לערך. אני מעביר את האובייקט שלי 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 במסתנים השונים אני מבצע הליך מיוחד שנקרא 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 לצורך גם `prefix` וגם `name`.

במקום לעשות משהו זהה:

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

אני עושה משהו זהה:

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

זה בדוק אותו הדבר.

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

תרגיל:

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

Happy Birthday! You are X years old!

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

פתרון:

הקומפוננטה נשמרה בשם `xs.js` 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;
```

`xs.js` היא תיקרא כל:

`<Birthday age={10} />`

אפשר לראות שהדבר היחיד שמעניין כאן הוא שיש לי `props` ומהם אני לוקח את ה-`age`. כשאני קורא לקומפוננטה זו, אני מעביר לה `age`, בדוק כפי שלמדנו.

תרגיל:

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` אמורה להיראות כך:

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

פתרון:

:Birthday קומפוננטה

```
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.

תרגיל:

הארქיטקט 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 קומפוננטת Birthday על מנת לעבוד עם ה-API החדש.

פתרונות:

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

```
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;
```

תרגיל:

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

פתרון:

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

```
const phrase = 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 בשתנה אחת כדי ליצור משתנה אחר ולשים אותו במה שהקומפוננטה מחזירה:

```
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-XJS ולסינטקס המבלבל – זה בסופו של דבר ג'אווהסקרייפט.

פרק 8

דיבאת



דיבאג

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

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

לכרום <http://bit.ly/reactdevtool>
או על:
<https://addons.mozilla.org/en-US/firefox/addon/react-devtools/>

Home > Extensions > React Developer Tools



React Developer Tools

Offered by: Facebook

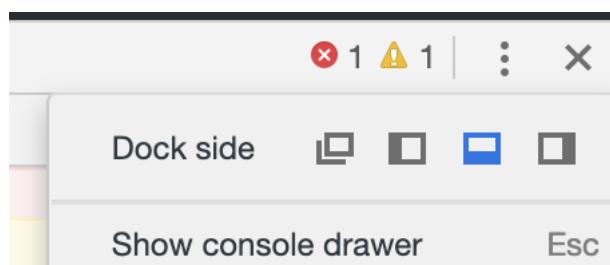
★★★★★ 1,195 | [Developer Tools](#) | 1,833,536 users

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

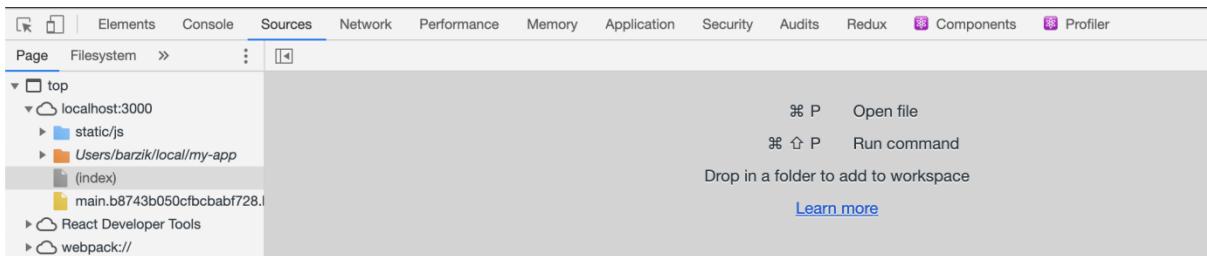
ראשית, הדיבאגר של הדפדפן עובד כרגע. בכל דפדפן מודרני יש כלי מפתחים מובנה ובפרק זה נלמד על כרום, הדפדפן הנפוץ היום. אך יש לנו מפתחים זהה בכל דפדפן עם אותו מבנה. בכרום אנו פותחים את כל הمبرקים באמצעות צירוף המKeySpecים קונטROL, שיפט ו-ו-בחלונות או בלינוקס וקומנד, אופשן ו-ו במק. יפתח לכם כל המפתחים בתחום המסך.

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

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



נעבור לתגית `sources`:



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

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

```

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

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

The screenshot shows the React DevTools interface with two tabs open: `App.js` and `Birthday.jsx`. In `Birthday.jsx`, a breakpoint is set at line 5. The code is as follows:

```

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

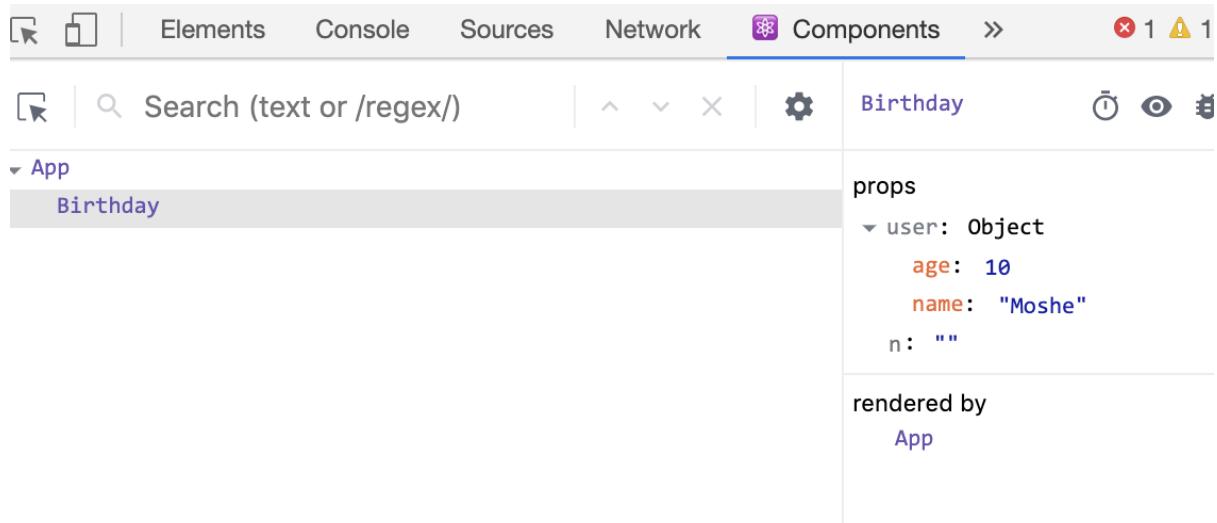
The debugger sidebar indicates the application is "Paused on breakpoint". The "Scope" section shows the following variables:

- Local**: age: 10, name: "Moshe", phrase: undefined
- Props**: {user: {...}}
- This**: undefined
- _props\$user**: {name: "Moshe", age: 10}
- Closure**: `(./src/Birthday.jsx)`
- Global**: (empty)

The "Breakpoints" section shows two checked breakpoints: one at `App.js:12` and another at `App.js:15`.

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

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



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

להרבה מפתחים מתחילה יש נטייה להימנע מהדיבagog כיון שמדובר בכלים שנראים מפחידים גם כריש המון מה ללמידה – אבל זה מצער. כדאי לשאקו כמה שעות בניסיון להבין איך הוא עובד ולהשתמש בו במקומות לנסوت לפענה מה הבעייה בעצמכם או בעזרתו כל מיני מעקפים כמו `console.log`. זה הרבה יותר קל ומהיר, ובטוח האורך – משתלם מבחינת הזמן. כלים המפתחים משתכליים ומשתפרים כל הזמן, וכך גם התוסף של ריאקט. יש סיכוי שבזמן שתקראו את הפרק, התצוגה של התוסף או של כלים המפתחים תשתנה – הם עוברים שינויים כל הזמן. אבל הפונקציונליות הבסיסית של כלים המפתחים לא משתנה – מקום לשימוש נקודת עצירה, להרייצ' קדימה ולראות מה מצב ה-`props`. אלו הדברים החשובים לכם.

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

פרק 9

סְנִירִים



סטוּטִיט

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

אדגנים את הבעיה עם קומפוננטה שנקראת `CountUp.js`. זוהי קומפוננטה שעשויה פשוט מאוד - היא מציגה את מספר השניות שאתה נמצאת באתר.

איך אני ממשש דבר זהה בג'אווסהקייפט? بكل קלות עם `setInterval`, פונקציה שאמורה להיות מוכרת למתכנת ג'אווסהקייפט ורצה כל פרק זמן מסוים במיילישניות. קוד שraz בכל שנייה נראה כך:

```
let i = 0;

function timeOutExample() {
  i = i + 1;
  console.log(i);
}

setInterval(timeOutExample, 1000);
```

לא משאו מרכיב מדי. פונקציית `timeOutExample` מקפיצה את `i` באחד ומציג אותה בקונסולה. עם `setInterval` אני מעביר את הפונקציה הזו ואומר שהיא תרוץ בכל שנייה (1,000 מיילישניות בכל שנייה).

אני יכול להכניס את הקוד הזה בקומפוננטה באופן הבא:

```
import React from 'react';

function CountUp() {

  let i = 0;
  function timeOutExample() {
    i = i + 1;
  }

  setInterval(timeOutExample, 1000);

  return (
    <span>{i}</span>
  );
}

export default CountUp;
```

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

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

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

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

export default App;
```

אני רואה שאכן הקומפוננטה מתרננדת ומוצגת הספרה 0, אך הספרה לא מתקדמת. אם תציגו בדייגר, שעליו למדנו בפרק הקודם, תוכלו לראות שאכן הפונקציה `timeOutExample` נקראת בכל שנייה, 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 שריינט מכניסה ומציגה אותו ב-DOM האמיתי. פועלות הציגה זו, שמתרגמת את JSX לציגות HTML שהדף יודע לטפל בה, נקראת "רנדור" מלשון `render` – הרצה. הרנדור הראשוני של כל קומפוננטה מתבצע בטיענה הראשונית ואנו רואים את התוצאה שלו בדף. כל עוד אין רנדור חדש, התציגה בקומפוננטה לא תשנה. זה בדוק מה שקרה עם

הקומפוננטה `CountUp`. ה-`setInterval` מעדכן ומעדכן את המשתנה הפנימי, אבל כל עוד אין רנדור מחדש, לא נראה את זה בדפדפן גם אם המשתנה הפנימי השתנה. על מנת להציג את השינויים שנעשו בקומפוננטה אלו חייבים לבצע רנדור מחדש. הרנדור חדש נעשה באמצעות אחת משתי דרכים:

1. שינוי חיצוני של `props`, ככלומר קומפוננטת אב משנה את ה-`props` של הקומפוננטה. שינוי זה גורם לרנדור מחדש מייד. בהמשך נלמד איך עושים את זה עם אירועים.
2. שינוי הסטייט של הקומפוננטה.

ברגע שיש רנדור מחדש, הקומפוננטה מתעדכנת וגם ה-XJS, ואנו רואים את הערכים החדשניים ב-`XJS`. בדוגמה שלנו, אם אנו רוצים שהקומפוננטה תציג את הערך של `z` שהשתנה על ידי `setInterval`, אנחנו צריכים לדאוג לשנות את הסטייט הפנימי שלה. בשביל זה אנחנו צריכים לדעת מה הוא סטייט.

סטייט

סטייט הוא בעצם הזיכרון הפנימי של הקומפוננטה. לקומפוננטות שעבדנו עד כה לא היה זיכרון. או שהן קיבלו את המידע שלהם מבפנים (קומפוננטות תצוגה) או שהן קיבלו את המידע שלהם מקומפוננטה אב דרך ה-`props`. אך או אחרת, הן קיבלו את הקטלט, עשו אליו חישוב והעיבו אותו ל-XJS. הקומפוננטה רונדרה, ה-XJS הוצג בתור HTML ובעצם הסיפורו הסטייטים. פה אנו זוקקים לזכור פנימי שיאפשר לקומפוננטה לנחל את המשתנים שלה, וחשוב מכך – לסמן לריאקט متى לבדוק לרנדור את הקומפוננטה מחדש. הסטייט במקרה של ריאקט הוא אובייקט שאנו מכנים אליו את המידע שאנו רוצים "לזכור". כאשר אנו משנים את המידע שיש בזיכרון, יש לנו רנדור מחדש.

از איך מנהלים את הסטייט? בקומפוננטות מסווג פונקציות אלו משתמשים במנגןון פשוט שנקרא `hooks` או ביחיד הוק. בהוק אנו מבצעים שתי פעולות:

1. הגדרת החלק שאנו מכניםים לסטייט.
2. הגדרת הפונקציה שבמיצועה אנו משנים את החלק הזה בסטייט. בקריאה לפונקציה זו מתבצע הרנדור מחדש.

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

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

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

function CountUp() {

  const [i, setI] = useState(0);

  function timeOutExample() {
    setI(i + 1);
  }

  setInterval(timeOutExample, 1000);

  return (
    <span>{i}</span>
  );
}

export default CountUp;
```

ראשית, אני קורא להוקים באמצעות `import`. ה-`import` הזה משתמש ב-`destructuring` שלמדונו עליון בפרקם קודמים. קבלו אותו כמוות שהוא - בכלים אלה צריך להשתמש כאשר אנו נעזרים בהוקים.

הצעד השני הוא, כפי שהסבירנו קודם, להשתמש בפונקציית `useState`. אני מעביר לה את הערך ההתחלתי (0) ומתקבל מערך. האיבר הראשון הוא קבוע המיצג את ערך הסטיטו, האיבר השני הוא

הפונקציה שמשנה אותו. מקובל (אך לא חובה) לכתוב את שם הפונקציה כך שיתחיל ב-set ואז יבוא שם הסטייט באות גדולות.

למשל, אם המשתנה שרציתי להכניס לזכרונו הוא count, שם הפונקציה שמשנה את המשתנה בזיכרון של הקומפוננטה יהיה setCount. במקרה שלנו שם המשתנה הוא אוט אוחט: i, אך שם הפונקציה שמשנה את המשתנה בזיכרון יהיה setI. זה הכל.
אני מבצע את ההגדלה זו באמצעות השורה:

```
const [i, setI] = useState(0);
```

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

עכשו אני צריך לשנות בקומפוננטה ולהמיר את כל הפעמים שבהן אני משנה את הערך של i בפונקציה של set, הפונקציה שאומרת בעצם – תכניס לזכרונו את הערך הזה, במקרה שלנו – 1 + :

```
setI(i + 1);
```

זה הכל!

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

שם הפעולה	הקוד החדש	הקוד הישן
להביא את ההוקים	<pre>import React, { useState } from 'react';</pre>	<pre>import React from 'react';</pre>
לקבוע ערך ראשוני לסטטייט, ולקבל את המשתנה לזכרונו, לקבל פונקציה שמשנה אותו לשנות את הערך	<pre>const [i, setI] = useState(0);</pre>	<pre>let i = 0;</pre>
	<pre>setI(i + 1);</pre>	<pre>i = i + 1;</pre>

הבה נדגים בדרך אחרת. ניצור קומפוננטה אינטראקטיבית שבה יש כפתור. המטרה שלנו היא להציג את מספר הפעמים שהמשתמש לחץ על כפתור מסוים.

נשמע מפיח? לא ממש. כפתור הוא אלמנט HTML פשוט, שלו נצמיד `onClick` שמא的动作 פונקציה שבה המונה משתנה. ניצור את הקומפוננטה באופן רגיל ופשוט:

```
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: הפונקציה שאנו מפעילים ב-`onClick` מוקפת גם היא בסוגרים מסולסלים כיון שאנו רוצים שם ששם יתורגם לקוד. אנו נמצאים ב-XJS וצריכים להבהיר שמדובר בפונקציה של הקומפוננטה.

אם תציבו את הקוד הזה ב-`CountButton.js` ותקראו לקומפוננטה מתוך `app.js`, תראו שלא משנה כמה פעמים אתם לוחצים על הכפתור, המספר 0 יישאר על הלוח. למה? כי הקומפוננטה

לא עברה רנדור מחדש. נכון, הקליק שינה את המשתנה הפנימי, אבל כל עוד אין רנדור מחדש, אנו רואים את תוצאת הרנדור הקודם גם אם ה-`count` משתנה.

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

1. לבצע `import` להוק הנוכחי, במקרה זה `useState`.
2. לומר לקומפוננטה בהתחלה أيיה משתנה נכנס לזכרון (`count` במקרה שלנו) ולהגדיר פונקציה שתשנה את המשתנה בזיכרון. שם הפונקציה הוא `set` וכך שם המשתנה מתחילה באות גדולה.
3. לשנות את הקומפוננטה שצריך באמצעות הפונקציה שמשנה את המשתנה.

```
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;
```

זה כל מה שצריך לעשות.

חשוב לציין כי לא כדאי להשתמש בסטייט בלי צורך. הסטייט וניהולו הם חלק חשוב מאוד בריект ואני Learned בהמשך על דרכי שונות לניהול סטייט גלובלי. בנוסף על כן, Learned גם על עוד הוקים.

תרגיל:

צרו קומפוננטה בשם CountDown.jsx שמקבלת כ-props מספר. הקומפוננטה תספור מהמספר זהה לאחר מכן (אין צורך לעצור ב-0).

פתרון:

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

function CountDown(props) {

  const [count, setCount] = useState(props.count);

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

  setInterval(timeOutExample, 1000);

  return (
    <span>{count}</span>
  );
}

export default CountDown;
```

הקומפוננטה זו עושה בדיקות אוטומטיות כמו הקומפוננטה של `CountUp` שיצרנו בפרק, אבל במקומות להעלות את `count` ב-1 היא מורידה אותו ב-1. גם פה כדאי לשום לב לשלוות החלקים:

1. ה-`import` השונה (מייבאים את ההוק של `useState`).
- 2.קובעים את המשתנה `shycnns` לזכור של הסטיטוס ואת שם הפונקציה שמכניסה את המשתנה לזכרון. שם הפונקציה תמיד מרכיב `set` ושם המשתנה.
3. כשרוצים לשנות את המשתנה, משתמשים תמיד בשם הפונקציה שמכניסה את המשתנה לזכרון.

תרגיל:

בקומפוננטה הקודמת שיצרתם, צרו קומפוננטה שתיפסק ב-0.

פתרון:

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

function CountDown(props) {

  const [count, setCount] = useState(props.count);

  function timeOutExample() {
    if(count !== 0) {
      setCount(count - 1);
    }
  }

  setInterval(timeOutExample, 1000);

  return (
    <span>{count}</span>
  );
}

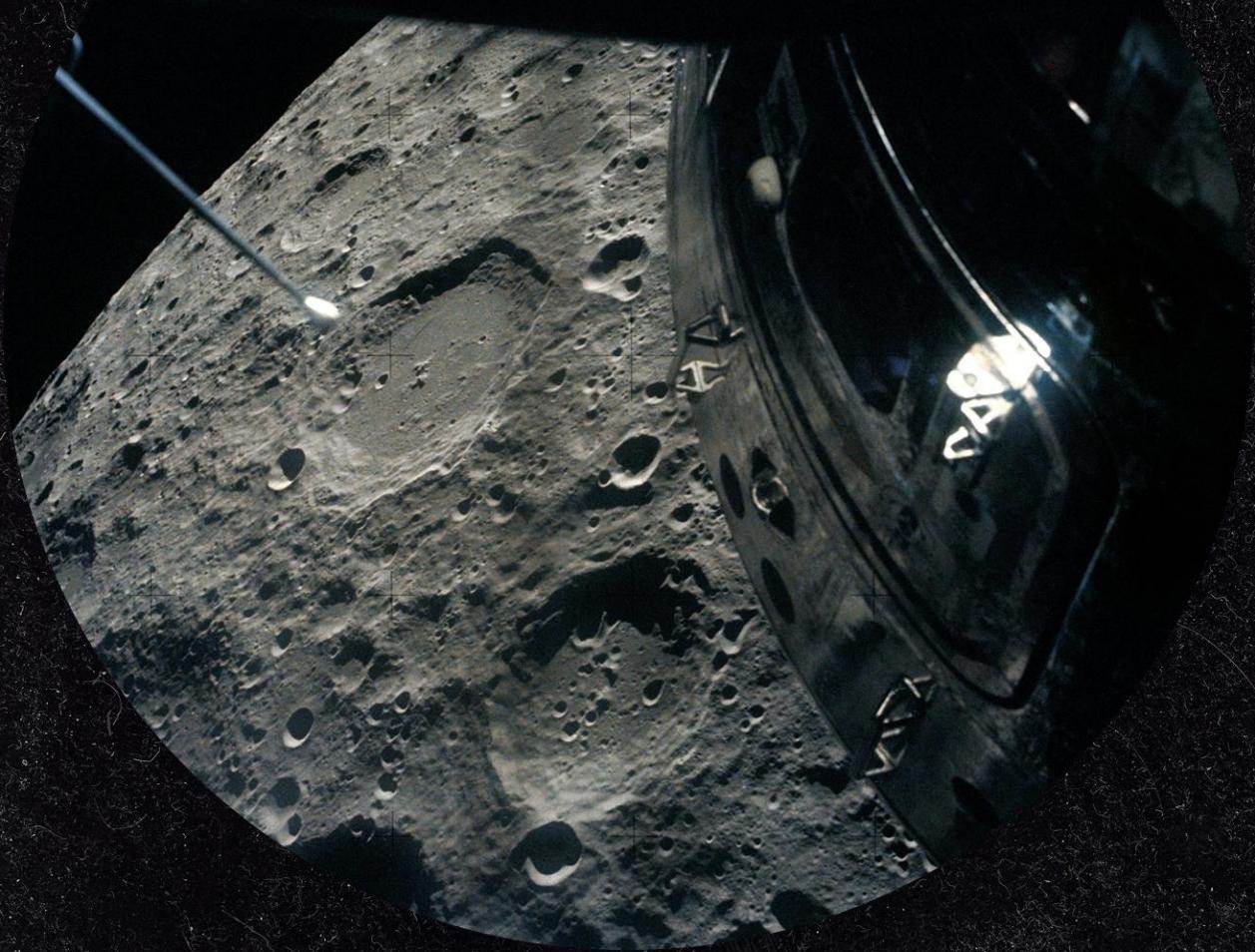
export default CountDown;
```

הפתרון כמעט זהה לפתרון הקודם. המטרה היא להבין שהסטיטיים אינם תורה מסיני. הם פשוט משהו שמותים בזיכרון, זה הכל, ואם לא משנים להם את הערך אפשר לגשת אליהם כרגע.

שימוש לב: מקובל למחוק את ה-interval אחרי שהוא מתאפשר, אבל זה לא מוקד התרגיל. במקרה חובה לבצע את זה כדי למנוע זליגת זיכרון.

פרק 10

תכננו מבנה הكومפוננטות



תכנון מבנה הקומפוננטות

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

הבה נניח שיש לי קומפוננטה שהקלט שלה (כלומר מה שהיא מקבלת ב-props) הוא מילישניות וഫטט שלה הוא זמן מעוצב בצורה נאה וקראית. שם הקומפוננטה הוא `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 אצוור קומפוננטה אחרת שמעבירה אליה את הזמן שאני רוצה שקומפוננטת הבת תציג. למשל, קומפוננטת `TodayTime.jsx` שמעבירה את הזמן של היום אל קומפוננטת `Watch.jsx`:

```
import React from 'react';
import Watch from './Watch.jsx';

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

את קומפוננטת `TodayTime.jsx` אני שם בכל מקום שבו אני צריך את השעה הנוכחית. אבל אם אני אציב את הקומפוננטה זו במקום כלשהו – למשל `app.js`, הדף הראשי של אפליקציית הריאקט – אני אראה משהו ממש מרגיז:

```
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 תרגלתם – אתם כבר יודעים שהרנדור של הקומפוננטה רץ פעם אחת בלבד וועל מנת עדכן אותו אנחנו צריכים לשים `setInterval`. מה הבעיה? אני יכול לשים את ה-`setInterval` כמו בדוגמה הקודמת, אבל אני לא רוצה לשימוש אותו ב-`js.Watch`. מדוע? לערבב לוגיקה עם קומפוננטת תצוגה זה לא רעיון טוב.

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

את זה למדנו לעשות בפרק הקודם. עם הוק של סטייט ניצור סטייט ונעדכן אותו בכל שנייה:

```
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);

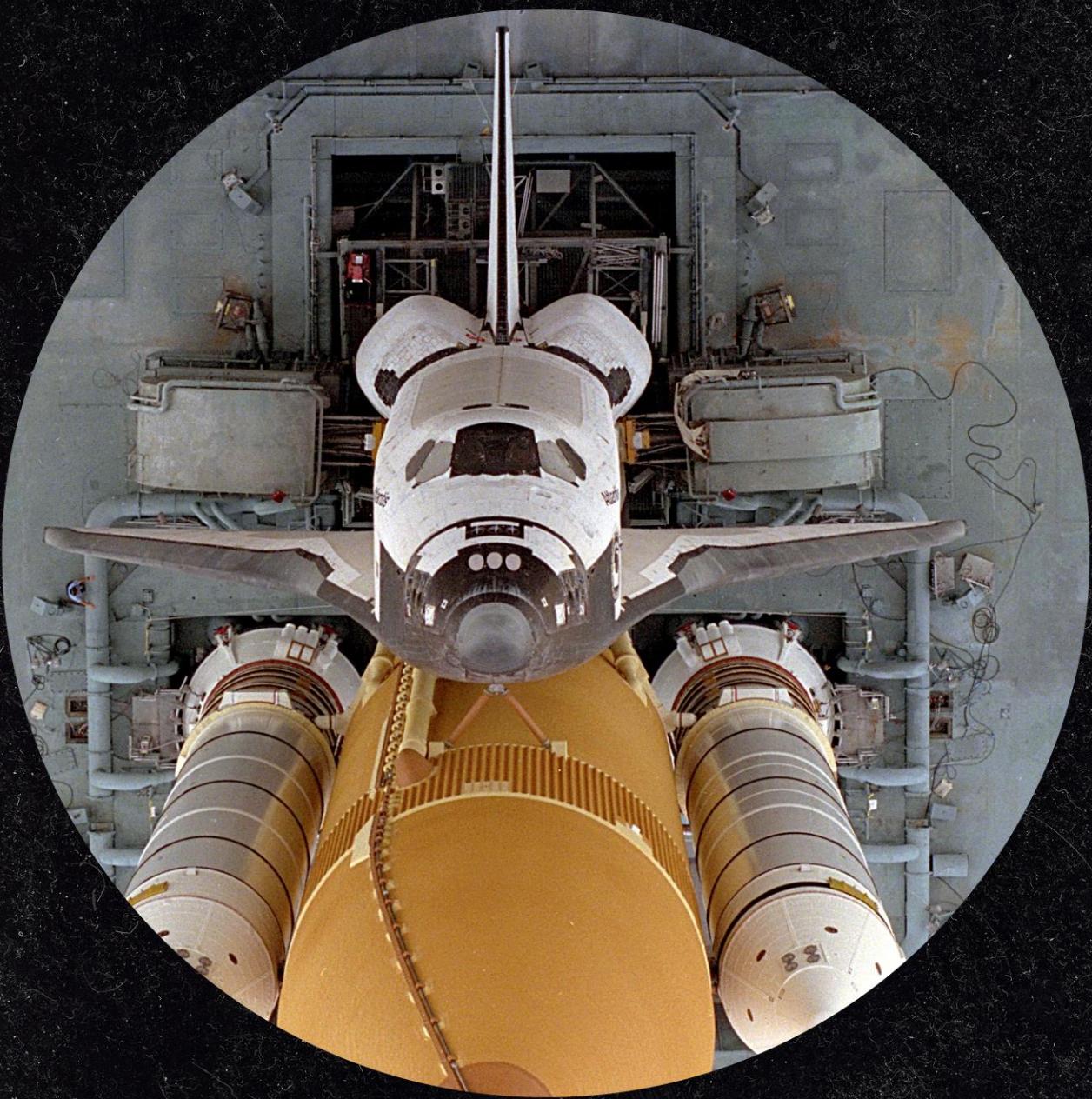
  return (
    <Watch milliseconds={today} />
  );
}

export default TodayTime;
```

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

פרק 11

אידועים ועדכון קומפוננטות



אירועים ועדכון קומפוננטות

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

אירועי DOM

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

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

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

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

אירועים סינטטיים/ ריאקטיביים

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

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

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

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

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

נדגים עם קומפוננטה פשוטה מאוד, שבה יש `input` שכל מה שנקליד בו יופיע על המסך שלנו. ניצור קומפוננטה שנקראת `MyInputViewer` ובה יהיה שני אלמנטים – `input` ו-`span` שייכיל את הטקסט. מה עם הטקסט? נשים אותו ב-`state` בבדיקה 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 מעבירים את המידע מתוך ה-`text` אל הסטיטוס `text`? בבדיקה בשבייל זה אנחנו משתמשים ב-`onChange`, שלפי שמו אנו רואים שהוא אירוע ריאקטיבי. איך הוא עובד? באמצעות תכונה שמצוידים לאלמנט `input`. התכונה זו מקבלת פונקציה שמופעלת כאשר האירוע מופעל. למשל:

```

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

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

e => console.log(e)

```

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

הקוד הזה למשל:

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

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

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

מופעלת. לא צריך להיבהל מהסינטקס המוזר. מדובר בפונקציית חץ שהיא חלק מג'אוועסקריפט. פונקציית החץ זו מקבלת אירוע `e` ומדפיסה אותו בקונסולה. אני ממליץ לכם לנסה 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 פחות מפחים פונקציית `cz`. נסו את הקוד הזה עכשו. הקליקו על ה-`button` ותראו איך פונקציית `clickHandler` מופעלת בכל לחיצה.

יש לנו הוסטירופט 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 מעבירים את מה שאנו מקלידים בתוך ה-`input` אל הסטייט?

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

```
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 הזו והקלידו בשדה ה-`input`. תוכלו לראות שהtekסט מופיע מיד ב-`span`. מדוע אנחנו מעבירים אל הסטייט רק את `e.target.value`? כי זה החלק באובייקט האירוע שכאמרור תכונת האירוע מעבירה לנו, שמניל את המידע שמעוניין אותנו. באובייקט האירוע יש

מידע רב ואמת מוזמנים לבחון אותו עם הקונסולה או עם הדיבאגר, אבל ב-`onChange`, המידע הטקסטואלי שיש בשדה מועבר עם `e.target.value`.

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

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

עכשו ניצור את הפונקציה ש יודעת לעבוד עם האירוע. במקרה הזה מדובר באירוע קליק שלא מכיל מידע, אז איך אני אדע לבדוק מה יש בתוך ה-`text`? טוב, זה קל – יש לי את אירוע `changeHandler` שמכניס כל מה שאני מקליד ישירות בשדה ה-`text` אל משתנה סטיטית שנקרא `text`. אני צריך רק לנתק את ה-`text` מה-`span` ולהברר אותו למשתנה סטיטית אחר (אקרא לו `viewText`), והליך תגרום לסטיטית `viewText` להתאכלה מסטיטית `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 שבסופו של דבר, האירועים בריאקט נראים למשתמש באופן כמעט זהה לאירועי DOM הטבעיים ולא צריך להיבהל מהם. אנו מגדירים את האירועים הריאקטיביים לאלמנטי HTML שתומכים בהם כמעט כמו האירועים של DOM, אך בדומה הבדלים קלים - שם הפקנץיה שונה (checkbox בעטיפה הריאקטית, onclick באירוע הטבעי) וכמובן מה שהוא מעבירים (פונקציית JSX שעתופה ב-{} בריאקט ומחזירת טקסט באירוע הטבעי). אם היינו משתמשים בג'אוויסקורייפט רגיל, הפעלת האירוע הייתה נראה כך:

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

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

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

```
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 בראקט, במיוחד שדות קלט, בקומפוננטה עצמאית. אנו עושים את זה פעמים רבות כי בקומפוננטות המכילות עיצוב, יש גם עיצוב מיוחד בקומפוננטה שימושי על ה-`input` או על ה-`button`, אז זה לא כל כך מופרך.

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

```
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 תטעלו ותנסו את הדוגמה בעצמכם, ככלומר תעתייקו את `Input` או `InputViewer` מkomponent `Input.js` או `InputViewer.js` מkomponent `App.js`. Create React App שלכם ב-`src`, תראו שהוא לא עובד. למה? כי komponent ריאקט לא מקבלות אירוזעים באופן טבעי.

از מה עושים? פשוט מאד. בקומפוננטה שלנו אנו דואגים לקבל את הפונקציה שמטפלת באירוע מה-`props`, בדוק כמו כל משתנה או נתון אחר, ואז את מה שהוא 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-`Input` אני לוקח את ה-`props.onChange` ומעביר אותו לאלמנט ה-`input` הטבעי. כך, אם אני משתמש בקומפוננטה `Input` באופן הבא:

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

היא תעבור לי.
באו נדגים שוב עם הceptor באמצעות אחת הדוגמאות הקודמות. אני אצור קומפוננטת כפטור ריאקטיבית שעוטפת כפטור HTML רגיל. שוב, זה נשמע מאולץ, אבל זו פרקטיקה נפוצה מאוד בספריות ובאפליקציות, ואם יצא לכם לעבוד או לראות קוד אמיתי בריאקט, תוכלו לראות את זה.

קומponentת Button.jsx תיראה כך:

```
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` ו מעביר את זה הלאה, לאלמנט `button` הרגיל, של ה-HTML, שיודע 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 לימדנו לעבוד איתם יפה מאוד ושהואותם 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, בקומפוננטה ריאקטית אנחנו חייבים להתייחס לכל אירוע שהוא רצוי ב-props ולהעביר אותו אל האלמנט הטבעי.

תרגיל:

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

רמז: יש צורך להציג שני מנהלי אירועים: הראשון הוא onMouseOver, שמאפשר לסייע 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 (שיםו לב שאלמנט בו הוא אלמנט HTML טبعי שיכול לקבל אירועים) שני אירועים שונים. האחד, `MouseOver`, נכנס לפעולה כאשר העכבר נמצא מעל האלמנט, והשני, `MouseOut`, נכנס לפעולה כאשר העכבר יוצא ממנו. כל מה שנותר לי לעשות הוא ליצור את הפונקציות שפועלות בזמן שהאירוע מתרחש. הראשונה לוקחת סטייט שיצרת ומכניסה לתוכו את המילה `active` והשנייה מספסת אותו. אני מכניס את המשתנה `shish` בסטייט ועובד איתה.

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

תרגיל:

ארכיטקט המערך הביט בתרגיל הקודם וביקש שה-`Div` יהיה קומפוננט ריאקט. ככלומר שהקומפוננטה 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}>
        </Div>
        {activeText}
      </div>
  );
}

export default MyDivContainer;
```

צרו את קומפוננט `Div` ב-`xsj.Div` כדי שהקוד שלעיל יעבד.

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

פתרונות:

```
import 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 התהיליך ולכתוב שהוא זה:

```
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 ה-`props`, להעביר אותם למשתנים ולהציב אותם באירועים של האלמנט הטבעי:

```

16   return (
17     <div>
18       <Div
19         onMouseOver={mouseoverHandler}
20         onMouseOut={mouseoutHandler}>
21       </Div>
22       {activeText}
23     </div>
24   );
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     <My div
10    </div>
11  );
12 }
13

```

בצם בדרך זו הפקנציות המטפלות באירועים זוברות מאלמנט האב אל ה-`props` בקומפוננטה שלו, וברגע שהן ב-`props` אני יכול לשים אותן באלמנט שיש בקומפוננטה הפונה אליו.

תרגיל:

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

פתרונות:

```
import 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 להציג אליו אירוע פשוט כמו בקוד `ג'אומajscriپט` רגיל.

פרק 12

אלמננט FRAGMENT



אלמנט **fragment**

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

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

תקבלו הודעת שגיאה:

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

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

```
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 לעטוף אותן ב-`div` 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. המונ פעים אני גם לא רוצה להכניס `div` או `span` כי אני לא יודע בתוך איזו קומפוננטה תוצב הקומפוננטה שלי וזה עלול להיות בעייתי. כמה טוב היה אם היינו יכולים להכניס אלמנט אב "ש��וף"! אז כן, יש אלמנט זהה שנקרא `React.Fragment`. האלמנט הזה יכול להיות במקום אלמנט אב והוא לא יודפס כלל. החלט 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 מתייחסים אליו. מקובל מאוד להשתמש בו בקומפוננטות בסיס ובקומפוננטות תצוגה.

פרק 13

USEFFECTS



useEffects

כפי שלמדנו בפרק על הסטייט, כשהקומפוננטה מסוימת משתנה, למשל בגלל אירוע מסוים, היא מבנית מחדש על התצוגה שלה. הבנייה זו מוחדש נקרואת `רנדור`. הבה נציג באמצעות קומפוננטה שנבנתה באחד התרגילים בפרק על אירועים – הקומפוננטה `Counter.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 שמשפיע על התצוגה של הקומפוננטה זו (במקרה זה עם כפתורים), אבל זה יכול להיות שינוי שנוצר כתוצאה מ-`props` שהשתנו), מתבצע רנדור מחדש. לא מעט פעמים אנו רוצים לדעת שהוא רנדור או רנדור מחדש של הקומפוננטה. בדיק בשביל זה אנו יכולים ליצור הוק מיוחד שיפעל פונקציה שאנו מעבירים לו.

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

הדרך הטובה ביותר להבין את זה היא באמצעות דוגמה:

```
import 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;
```

ראשית הבאתו את `useEffect` כhook ב-`import`. הכי קל ונחמד בעולם. השלב הבא הוא פשוט להשתמש בהוק זהה. אני קורא לו `useEffect` ומעביר בו ארגומנט שהוא הפונקציה שתפעול.

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

מה שיופיע הוא:

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

בכל פעם שהקומפוננטה תרוץ. נסו את זה! השתמשו בקומפוננטה זו והציגו בקונסולה של כל המפתחים כאשר אתם מושנים את הערכים 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 לבלבול, אבל צריך לזכור שהן בסופו של דבר פונקציות רגילות לכל דבר עם כמה תוספות חביבות, כמו למשל שמירה על `this`. זו הסיבה שכדי מאד להשתמש בהן.

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

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

```
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

קומפוננטה ללא שם



קומפוננטה ללא שם

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

```
import React from 'react';

function Button(props) {

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

export default Button;
```

קראונו לפונקציה `Button` ואז ייצאנו אותה. זה סינטקס וליידי שונה להבנה לאנשים שלומדים ריאקט לראשונה. אבל יש דרך נוספת לנוספת לכתוב פונקציות. בסופו של דבר, למי שמייבא את הפונקציות ומשתמש בהן אין שום צורך בשם הפנימי הזה והוא רק לצורך הנוחות שלנו. אבל גם אנחנו לא זקוקים לשם זהה. ה-`export` מקבל גם פונקציות חוץ אונומיות. ככלומר כל קומפוננטה שנכתבנו 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 מפחד, מבעית ואפיו מוזר לרוב המתכנתים שלא מרגלים בריאקט. אבל אם הגעתם לפרק זהה וקראתם היטב את כל הפרקים - וגם תרגלם - תוכלו להבין את הקוד הזה אם תיתקלו בו. מה שנעשה כאן הוא פשוט: במקום להציג על שם פונקציה, אני יוצר אותה כפונקציה אונומית ופשוט מחזיר אותה עם `export default`. הבעה המרכזית היא שכלי המפתחים של ריאקט לא יציג את שם הקומפוננטה ויקשה עליינו למצוא אותה.

פרק 15

NEYAOV KOMPOUNNOT



עיצוב קומפוננטות

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

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

ה-import כאן מבצע בפשטות ייבוא לשם הקובץ. בקובץ הראשי של אפליקציית הריאקט, app.js, אנו יכולים לראות את ה-import הזה:

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

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

כדי להבין איך CSS עובד ואיך הוא עובד בפרויקט, אנו נעבד עם הקומפוננטה `Counter.js`. שבנינו בפרק על אירופים.

```
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 וזה ניבא אותו. אנו נשימים בראש הקומפוננטה את הטקסט:

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

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

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

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

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

CSS בסיסי

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

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

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

סלקטור

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

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

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

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

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

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

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

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

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

כז אנו יכול לבחור רק את ה-buttons שיש להם קלאס myDiv בואופן הבא:

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

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

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

```
<div id="myDivId">{count}</div>
از הסלקטור יהיה:
```

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

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

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

תכונות

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

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

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

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

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

הערכים שלhn	שם התכונה	תיאור התכונה
מספר + ak, למשל: 100px	height	גובה
מספר + ak, למשל: 100px	width	רוחב
שם הцבע, למשל red, או ערך הצבע בהקס' למשל: #fffffff	color	צבע טקסט
שם הцבע, למשל red, או ערך הצבע בהקס' למשל: #fffffff	background-color	צבע רקע
מספר + ak ארבע פעים. הערך הראשון קובע את השוליות העליונים, השני את השוליות מימין, השלישי את התחתונים והרביעי את השוליות משמאל. למשל: .10px 10px 10px 10px	margin	שוליות
right או left	text-align	כיוון טקסט בתוך האלמנט

עכשו, כשייש לכם את היסודות לבניית CSS בסיסי, תוכלו ליצור את קובצי ה-CSS שלכם ולבצע להם import או לחלופין לעצב את הקומפוננטה בדרכים אחרות. יש כמה דרכים נוספות לעיצוב קומפוננטה בריאקט, אבל כל הדרכים האלה נשענות על ספריות צד שלישי. לריאקט עצמה אין דעה בנוגע לדרכי העיצוב, ובוודוקומנטציה 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

ללאס ריאקטוי



קלאס ריאקטיבי

עד כה, כל הקומפוננטות שעבדנו עלייהן היו פונקציות. הקומפוננטות האלו נקראות באנגלית **functional component** והן משתמשות בהוקים על מנת לנצל את הסטייט וגם לדעת מתי הקומפוננטה רונדרה. אבל יש גם קומפוננטות אחרות, קומפוננטות מבוססות **קלאס**.

אם הספר הזה היה נכתב בתקופה שריאקט הייתה בגרסה 15, סביר להניח שהקומפוננטות האלו היו תופסות בו נפח גדול יותר, אבל החל מגרסה 16.8 השימוש בקומפוננטות מבוססות פונקציות הפך לקל יותר. כרגע אפשר לראות מעבר של יותר ויותר מפתחים ושל ריאקט עצמה לכיוון הקומפוננטות מבוססות הֆונקציה. אבל חשוב לדעת שיש קומפוננטות מבוססות קלאס בריאקט ولو רק בגלל היכולת שלכם לעבוד עם קוד ישן יותר (Legacy Code). נוסף על כן, עדין יש שליטה טובה יותר במעגל החיים הריאקטיבי בקלאסים, אבל הדגש הוא על עדין.

از איך זה נראה? פשוט מאד – עם קלאס. קלאס (Class) הוא חלק מג'אוועסקרייפט עצמה ומדובר בסינטקס שונה מפונקציה. לקלאס יש כמה חלקיים – ראשית, יש לו קונסטרוקטור (**constructor**), הֆונקציה הבנאית, שפועלת מיד בהתחלת ושאליה אני גם מקבל את ה-**props**. הקוד בקונסטרוקטור מתבצע ברגע הראשון בלבד. שנית, יש לו **render**, שיש בו מה שהקומפוננטה מחריצה.

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

```
import React from 'react';

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

export default Welcome;
```

המקבילה שלה בקלאס 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 כיון שהקלאס שלנו ירוש מה-`React.Component`, אחראית על החזרת ה-`JSX`. עד כאן אין הבדל משמעותי בין קומפוננטת קלאס לקומפוננטת פונקציה, שאוותה אנו מכירים.

העניין נותרים פשוטים גם אם אנו מתחילה להשתמש ב-`props`. ה-`props` בקלאס נמצאים על גבי הסkop של הקומפוננטה, כלומר ב-`this.props`. בכל מקום נתון בקומפוננטת הקלאס אני יכול להשתמש ב-`props` ולהשתמש ב-`this.props` ולקבל גישה ל-`this.props`.

כדי להדגים, הבה נניח שאנו רוצים להוסיף את שם המשתמש ולקבל אותו מה-props. בקומפוננטת פונקציה נעשה שהוא זה:

```
import React from 'react';
function Welcome(props) {

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

export default Welcome;
```

בכלאש אנו ניגשים אל ה-props באמצעות ה-this. אבל למעט העובדה הזאת, אין שינוי מהותי. על מנת למשם את אותה התנהגות בדיק עם קלאס, 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;
```

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

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

```
import 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 זה נדרש להשתמש בקונסטרוקטור, שבו נאתחל את הסטייט ונכנסים לתוכו ערך ראשוני. הסטייט הוא אובייקט שיושב בסkop של הקומפוננטה (כלומר תחת `this`). אין לנו הוק במקרה זה 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 קריטית כיון שהיא קוראת לكونסטרוקטור של `React.Component`. זו לא המלצה של ריאקט אלא חלק מהסינטקס של ג'אווהסקריפט. בלי השורה הזו, הקונסטרוקטור שלכם לא יעבד כשרה.

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

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

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

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

הבה נדגים על ידי המרת הקומפוננטה הזו, שהיא פונקציה 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 לקלאס, ראשית נזכיר ליצור קונסטרקטור ולשנות את הסטיטייםם, אבל יש

לנו גם אירועים, ולאירועים אלו חיבום להעביר קונטקסט של `this`. אחרת, בשעת הפעלת האירוע תעבוד הפונקציה בסКОפ אחר (הסקופ של החלון) ולא תהיה לה גישה לסטיטו.

```
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 לשימוש 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 אומדן? מסיבה פשוטה מאוד – אם האירוע מופעל, למשל האירוע זהה:

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

הסקופ הוא לא של הקלאס. על מנת לגרום לאירוע לעבוד בתחום הקלאס עם גישות למשתנים, למתודות ותכובות לסטיטיט, יש צורך להשתמש ב-bind. ה-bindגורם לאירוע לעבוד בסקופ של הקלאס. אם ה-bind לא יהיה קיים, אני אקבל שגיאה כזו:
TypeError: Cannot read property 'setState' of undefined

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

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

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

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

מעגל החיים ב-**קלאס**

אחד היתרונות הגדולים של שימוש בקומפוננטה מבוססת קלאס על פניו פונקציה הוא יכולת השיליטה הגדולה יותר במעגל החיים שלה. בעוד בפונקציה יש לנו רק את הhook `useEffect`, שפועל כאשר הקומפוננטה מתրנדרת בפעם הראשונה וכאשר היא מתרנדרת מחדש, ב-**קלאס** יש לנו שליטה מלאה יותר במעגל החיים וKİימות לא פחות משם מתודות לעשות זאת.

componentDidMount

מתודה המופעלת מיד לאחר שהקומפוננטה נטענת לתוכה-**DOM**. זה מקום מעולה לבצע קריאה-**API** ופעולות נוספות הקשורות לאתחול הקומפוננטה (כמו לוגינג) או להירשם לאיורוועים, כפי שנראתה בדוגמה שבמהמשך הפרק.

componentDidUpdate (prevProps, prevState, snapshot)

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

שימוש לב: בנגד `componentDidMount`, שנקרא רק פעם אחת במעגל החיים של קומפוננטה, נקרא לאחר `componentDidUpdate` כל פעם שמתבצע רנדור (חוץ מהפעם הראשונה, שבה `componentDidMount` מטלף). כדאי גם לשים לב שטפני שהמתודה הזאת מופעלת כתוצאה מ-`setState`, אם נשים בתוכה עוד `setState` בלי תנאי, עלולה להיות לנו לולאה אינסופית.

componentWillUnmount

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

shouldComponentUpdate (nextProps, nextState)

מתודה שבה אנו יכולים לקבוע אם אנו בכלל רוצים שהקומפוננטה תתרנדר 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 זהה. היא נדירה יחסית ומאפשרת לנו גישה אל ה-**props** שהתעדכנו ואל הסטייט החדש. היא מתרכשת לפני `componentDidUpdate` וכך אמרו, אם היא

מחזירה `false`, אז `componentDidUpdate` לא תרצו כיון שהקומפוננטה לא תרונדר. התכנון העתידי של ריאקט הוא לאו דווקא להפוך את זה להוראה אחידה אלא להמלצה למנוע את הרנדור של ריאקט.

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

getDerivedStateFromError (error)

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

componentDidCatch (error, info)

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

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

הפונקציה `componentWillUnmount` בעצם מופעלת אוטומטית כאשר הקומפוננטה נמחקת מה-`DOM`, וזה קורה כתוצאה של פעולה שוננות, כמו מעבר דף בראוטינג, שעליון נרחב בשלב מאוחר יותר.

דוגמה טובה נוספת לשימוש במתודות של מיגל החווים היא למשל בתפיסת שגיאות. מקובל מאוד להשתמש בקומponeנטה אב צו, 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 בקומponeנטה מראה שהיא תופסת כל שגיאה ומציגה אותה. אם לא – היא מציגה את מה שיש בתוכנה. זה מקובל מאד ואת זה, נכון לגרסה האחרונה של ריאקט, אפשר לעשות רק בקלאס.

תרגיל:

צרו קומפוננטה מובוסת 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 מהهو שאינכם מכירם. את הסטייט אנו מגדירים כבר בקונסטרוקטור ואנו מתחלים אותו כ-0, ויצרנו handler לקליק שמעלה את הסטייט ב-1.
כיוון שהAIRORU פועל בסקוופ אחר, אנו חייבים לקשרו אליו את הסקוופ שלנו באמצעות:

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

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

תרגיל:

מנהל המוצר של בזק פנה אליכם בבקשתה ליצור קומפוננטה שמציגה התקפות סייבר. המספר שלתן מתחילה ב-1,000 ועולה ב-10 בכל שנייה. עלייכם למש这套 עם קלאס כיוון שתפקידם צריכים להשתמש ב-setInterval ויש צורך לנוקוט 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 אחרי שהקומפוננטה נעלמת.

ראשית, אנו משתמשים בקלאש כרגע. כיוון שיש לי צורך בזיכרון פנומי, אני יודע שאני צריך סטייט ואני מגדר סטייט כבר בקונסטרוקטור באמצעות `this.state`.

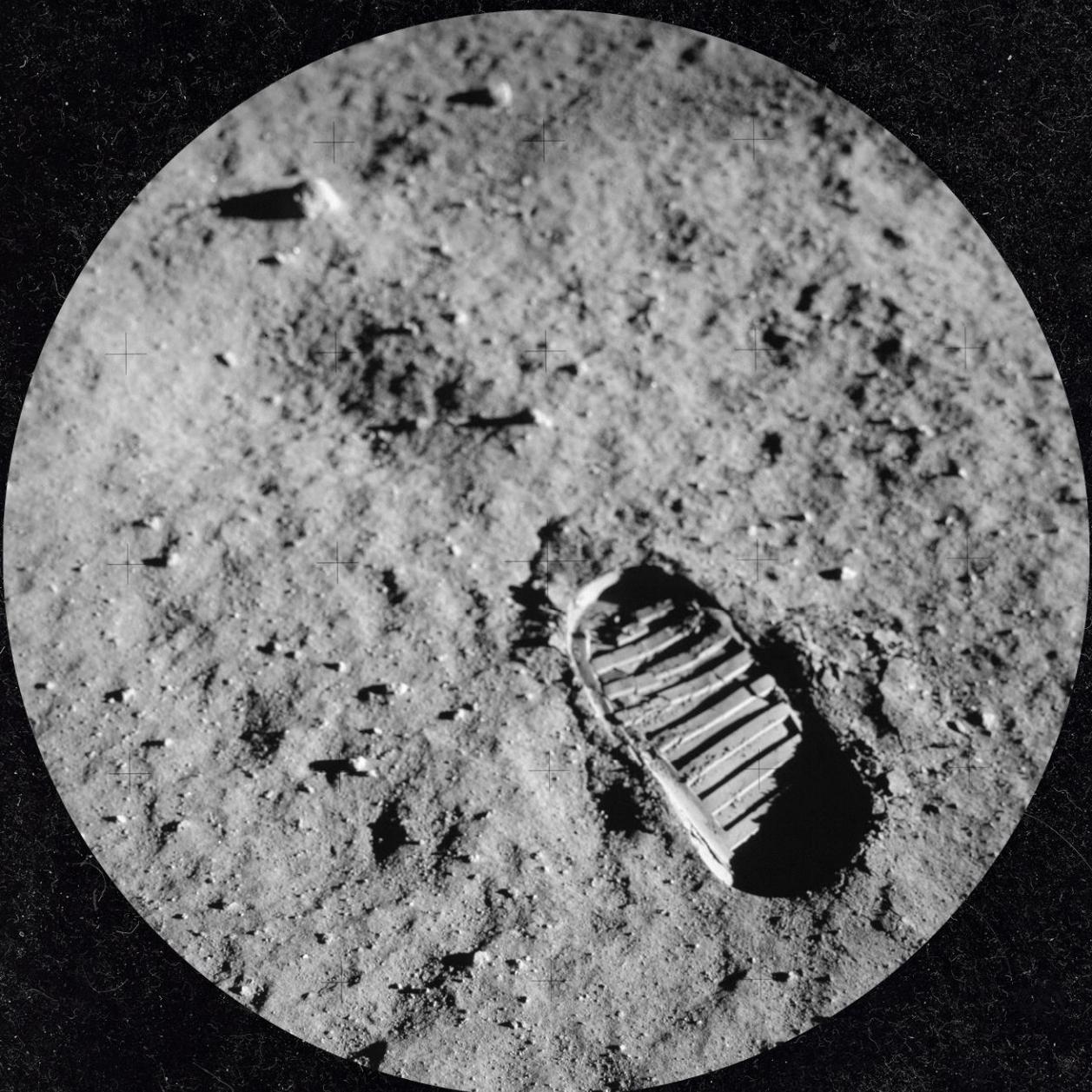
הפעולה השניה היא להגדיר `interval` שיפעל בכל שנייה, ואת זה אני עושה במתודת מעגל החיים `componentDidMount`, שעובדת מיד לאחר שהקומפוננטה מופעלת. אני מייצר `interval` ויוצר לו ופרנס במשתנה שנמצא בסkop של הקומפוננטה. מה שה-`interval` עושה הוא לעדכן את הסטייט ב-10+. על מנת שתהייה לו גישה לסטטייט, אני מפעיל:

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

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

פרק 17

שיכון בקומפוננטות סמליקות אחרים



שימוש בקומפוננטות אחרות

אחת מהחווקות הנדרשות של ריאקט היא שימוש בספריות אחרות של קומפוננטות. וכך בעצם אנחנו לא נדרשים לבנות מ一封 את האפליקציות שלנו אלא פשוט להשתמש בקומפוננטות בקוד פתוח שאחרים בנו וכך אנחנו יכולים להגיע לתוצאות טובות מאוד בזמן קצר יותר. יש בחוץ אינספור קומפוננטות וואוסף קומפוננטות שקל לנקח וליצור איתם אפליקציות מורכבות ביותר ויפות מאוד מבחינה ויזואלית.

ספריות האלו באוט כabilות תוכנה שאנו יכולים להתקין בפרויקט שלנו, שייצרנו ב-Create React App. הפרויקט הזה מכיל לא מעט כabilות תוכנה, כמו babel למשל, והabilities האלו מנהלות על ידי ניהול התוכנה של Node.js (ג'אוوهסקריפט בסביבת השרת) שנקרא webpack. לא נדרש לדעת Node.js כדי להשתמש ב-webpack ולהתקין כabilות תוכנה (או להסיר אותן). על מנת להתקין כabilית תוכנה הכוללת קומפוננטות, אנו נדרשים לשורת הפקודה שעליה למדנו בפרק על בניית סביבת עבודה מתקדמת. אנו נפתח את Visual Studio Code, נחפש את לשונית Terminal וナルץ על New Terminal ולבצע התקנה באמצעות webpack.

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

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

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

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

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

הפקודה הזו משתמשת ב-PLACEHOLDER על מנת להוריד ולהוסיף לפרויקט באופן אוטומטי את סדריית UI או React Material על הקומפוננטות שלה.

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

```
PS C:\Users\barzik\local\my-app> npm install @material-ui/core
npm WARN @typescript-eslint/eslint-plugin@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @typescript-eslint/parser@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN ts-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 || >= 3.6.0-dev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\chokidar\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\jest-haste-map\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.0.7 (node_modules\fsevents):
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@4.6.1
added 31 packages from 41 contributors and audited 903766 packages in 17.413s
```

עכשו אפשר להשתמש בקומפוננטות ה-PLACEHOLDER.

חשיבות: בשנים האחרונות הפכו js.js ו-PLACEHOLDER לשימוש נורמלי בכל הקשור לסביבות פיתוח וכל הסprüיות 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

```
npm install PACKAGE_NAME
```

תיאור הפעולה

התקנה של חבילת תוכנה

```
npm uninstall PACKAGE_NAME
```

הסרת התקנה של חבילת תוכנה

```
npm ls PACKAGE_NAME
```

בדיקה אם חבילת תוכנה מותקנת אצלכם

או

בдиוקה של הקובץ package.json

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

אחד הקומponeנטות היא קומponeנטת כפתור בשם Button, שבעצם יוצרת כפתורים מעוצבים. הנה נשתמש בה עם הקומponeנטה Counter שבנוינו 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 תרצו את הקומפוננטה זו, 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 בדיק 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 לבבל או להטריד בתחילת, אבל זה פשוט שם חביבה (אפשר להשתמש בתווים מיוחדים בשם חבילה). זה הכל!

בדוגמה אפשר לראות שהעברית `props` בשם `color` ו-`variant` משמשים על מראה הכפתורים. בטעוד של הקומפוננטה של הכפתור, וכל קומפוננטה אחרת, יש הסברים על ה-`props` שאפשר להעביר ועל אין הם משנים את המראה של הקומפוננטה.

אפשר, כמובן, להשתמש בכמה קומפוננטות שרצים 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;
```

ייבוא כמה קומפוננטות

אם אני מיבא כמה קומפוננטות מסוימת ספרייה, מקובל להשתמש בסינטקס בסיסי קצת שונה ליבוא.
במילים:

```
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'אוועסקריפט - זו צורת כתיב שונה לאותו הדבר.

חשוב לציין שהכח האמתי של ריאקט – השימוש בספריות קומפוננטות כאלה, שנונות
לנו כוח אדיר רק באמצעות התקנה פשוטה של ספריות קוד פתוח ו שימוש קל. באתר של React
ול Material יש דוגמאות והסבירים רבים כיצד להשתמש בקומפוננטות, ובזמן קצר מאוד אפשר
לבנות דפים מורכבים מאוד.

הבה נדגים עם קומפוננטה אחרת, שבאה על בסיס קומפוננטות אחרות: Grid. אפשר להציג באתר
הרשמי של הקומפוננטה – שם יש הוראות התקנה מדיוקנות:

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

אבל לנו נדגים גם פה – ראשית יש להתקין את הקומפוננטה באמצעות npm באופן הבא:

```
npm install @devexpress/dx-react-core
npm install @devexpress/dx-react-grid
npm install @devexpress/dx-react-grid-material-ui
npm install @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 לחסוך לכם עבודה ומאפשרים לכם לקבל תוצאות
 מהירות ויפות כמעט מייד. שווה להתחיל להתעניין ולנסות, למשל, את UI React Material. יש
 הרבה דמואים ודוגמאות קוד מעניינות שדרכם אתם יכולים ליצור תוצר מהם כמעט מיד עם עבודה.

מודולים חסרים

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

```
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 מוקח. במקרה הזה פשוט כדאי לנצל את התוצאה או להתקין את הספרייה. בשגיאה זו הספרייה החסורה היא `@material-ui/icons`

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

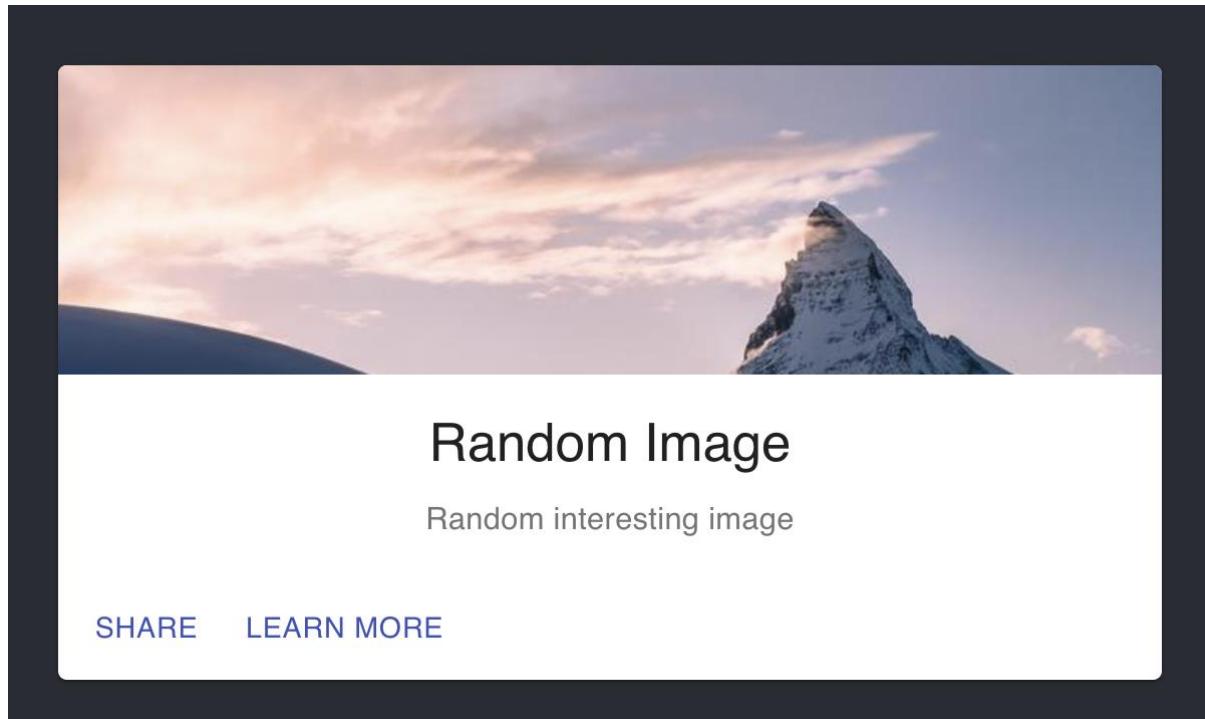
היא מתקינה את הספרייה החסורה שצורך להשתמש בה.

תרגיל:

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

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

צרו את הרכיב זהה:



רמז: התיעוד והדוגמאות לכרטיס נמצאים בכתובת הzon:

<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>
);
}

```

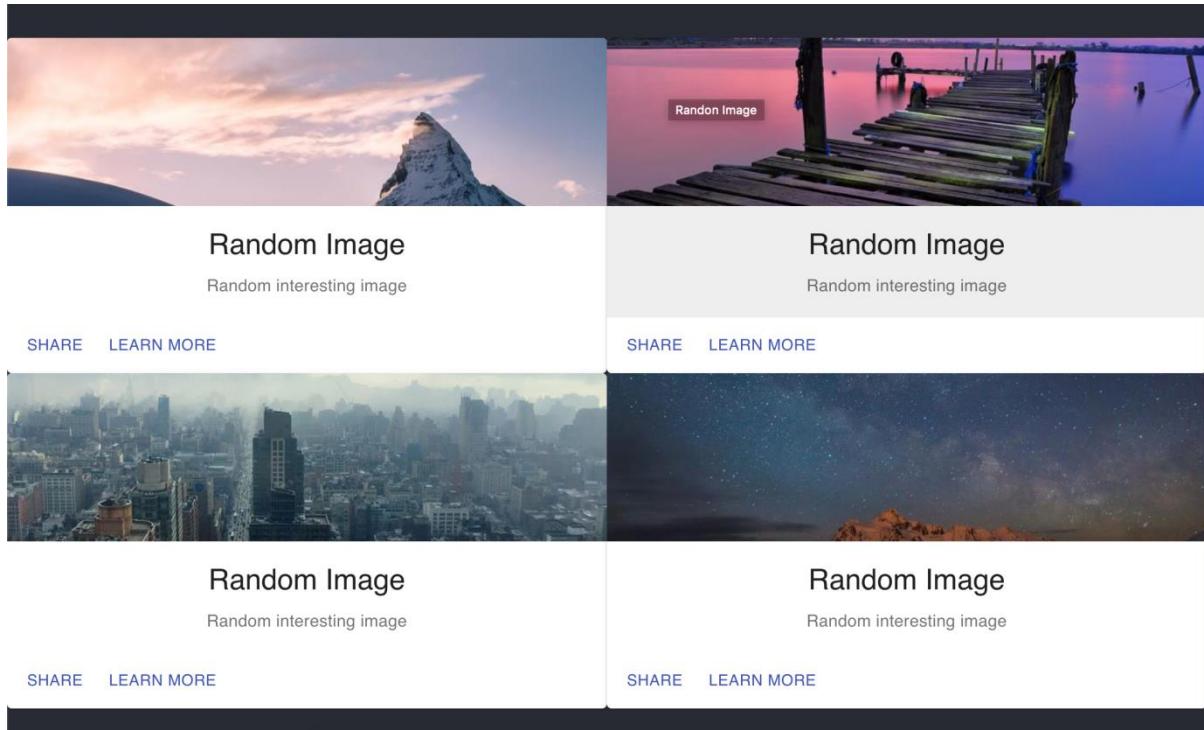
אין כאן גאונות גדולה, פשוט העתקתי את הדוגמה המופיעה בדוקומנטציה כדי ליצור את מה שרציתי ליצור. במקרים האלו אין מה לשבור את הראש ולהתאמץ, אלא צריך פשוט לקחת את מה שיש בדוקומנטציה.

כדי לשים לב שהדרך של UI React Material קבועה עיצוב היא דרך שונה המשמשת `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`, וזו להשתמש בקומפוננטה זו איפה שרצוים או לדאוג להעביר לה `props` שיקבעו את התמונה השונה.

תרגיל:

באמצעות הרכיב הקודם, צרו ארבעה כרטיסים כאלה:



עם התמונות שבקישורים הבאים:

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

<https://picsum.photos/id/867/700/400>

<https://picsum.photos/id/868/700/400>

<https://picsum.photos/id/869/700/400>

רמז: העבירו את התמונה לקומפוננטה הקודמת באמצעות props במאפשר `copy` שלמדנו.

פתרונות:

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

<CardMedia

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

אמנם הקומפוננטה נראה 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}>
    <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 לkomponenta את הפרמטרים האלו ולעשות את זה ארבע פעמים בkomponenta 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. תוכלו לבדוק אותה בדוקומנטציה. אם השתמשת ב-CSS זה מעולה, אבל קל יותר להשתמש בקומפוננטה מסודרת שהיא כבר יצר.

כל שנוטר לי הוא לחת את קומפוננטת MyContainer ולהציג אותה איפה שאני רוצה, למשל ב-`:app.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 מהרמים. לכל קומפוננטה וקומפוננטה יש API מסוימת. כך, למשל, ב-UI-UI Material אפשר להעביר `onClick` כדי לבצע פעולה מסוימת, כמו לפתוח טאב חדש.

תרגיל:

נוסף על פרמטר התמונה, העבירו גם פרמטר של קישור. לחיצה על תפוח את הקישור זהה.

רמז: פתיחת קישור בלשונית דפדף חדשה נעשית באמצעות:

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

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';
import CardContent 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="Randon 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 לפחות משפע הקומפוננטות שיש פה. יש כפתור? מצוין, אפשר להעביר אליו onClick, בדיקת כפי שלמדנו. הכפתור, שהוא רכיב של UI, React Material, יודע יפה מאוד לטפל באירוע. אנו מעבירים לו אירוע פשוט שפותח מה שאנו מעבירים לו בו-props. במקרה זהה:

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

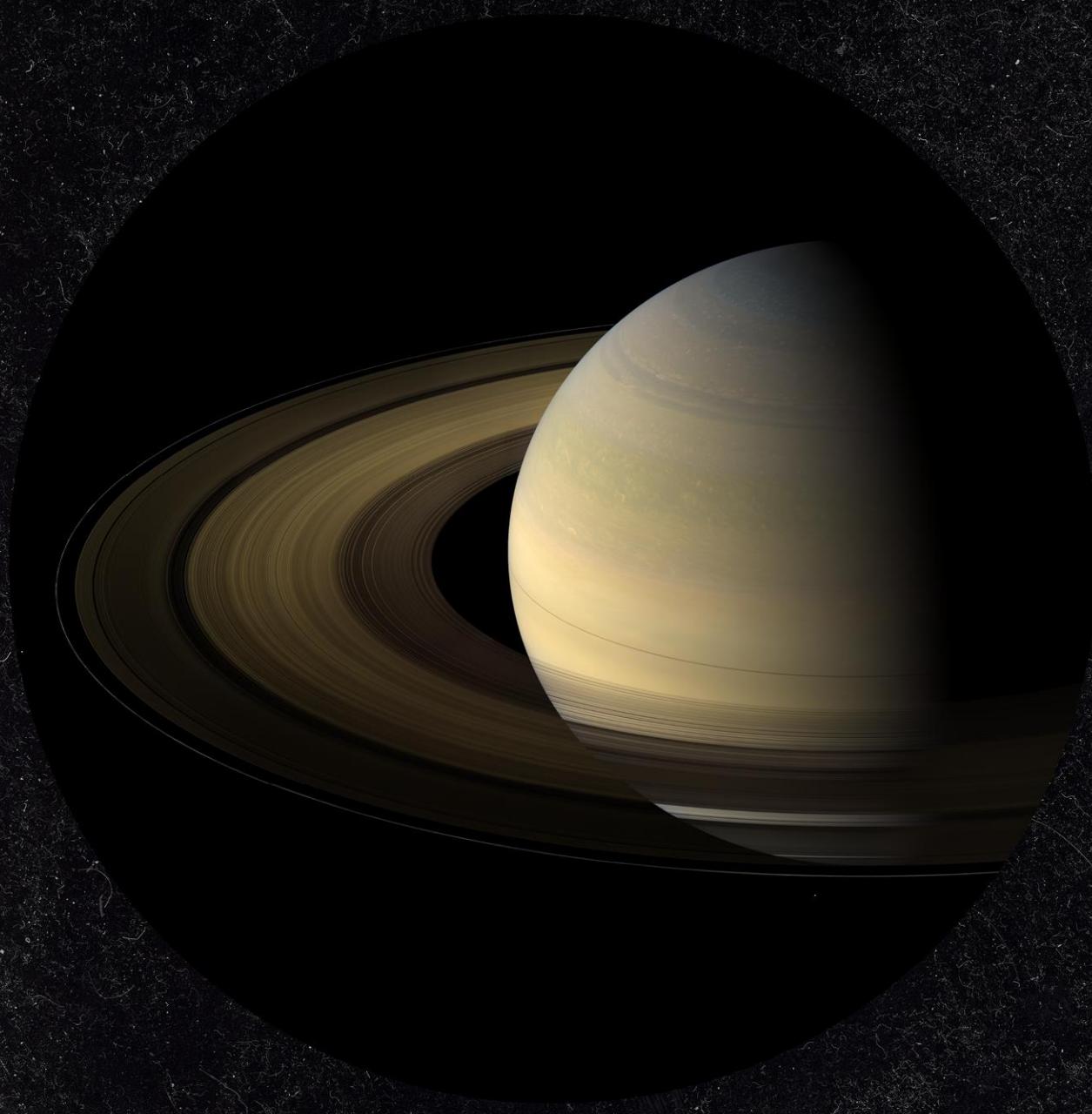
כל מה שעליינו לעשות הוא להעביר לקומפוננטה שלנו את הקישור. במקרה הזה הגדרתי שהפרמטר יהיה linkTo.

```
<Grid container item xs={12}>
  <RandomImageCard LinkTo="https://internet-israel.com"
    imageSrc="https://picsum.photos/id/866/700/400" />
  <RandomImageCard LinkTo="https://hebdevbook.com"
    imageSrc="https://picsum.photos/id/867/700/400" />
  <RandomImageCard LinkTo="https://he.wikipedia.org"
    imageSrc="https://picsum.photos/id/868/700/400" />
  <RandomImageCard LinkTo="https://haaretz.co.il"
    imageSrc="https://picsum.photos/id/869/700/400" />
</Grid>
```

כרגע אני יכול המשיך ולפתח על בסיס הקומפוננטה. שימוש לב שימוש, במקובל, במקומם 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 המועשת. העשרה זו מתבצעת באמצעות פונקציה עוטפת שמקבלת כารוגומנט קומפוננטה. הפונקציה שמבצעת את השינוי נקראת **HOC**, ראשי תיבות של **Higher Order Component**. המשמעות של המושג זהה היא "קומפוננטות מדרג גבוה" – כלומר ככלו שלא מילכילות את הידים בפונקציונליות אלא מנפקות קומפוננטות אחרות, שהן אלו שעשוות את העבודה.

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

כשאנו כותבים HOC, אנו צריכים לזכור שהקלט הוא קומפוננטה והפלט הוא קומפוננטה אחרת שמשתמש בהם והשימוש בהם עלול להיות מבלבל או סתום יכול להיראות כמו וודו – ככלומר CIS שמשתמש בקומפוננטה של הקלט כבסיס. 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 מעשי את הקומפוננטה? הבה נציגים עם קומפוננטה שאנו רוצה להעשייר אך ורק בתוכנה אחת שאנו רוצה להעביר לה. איזו? משמעות החיים, הקיום וכל

השאר. ערך התוכנה הזו, כפי שידוע כל מי שקרא את הספר "מדריך הטרטמפיקט לגלקסיה", הוא 42. איך אני אוסיף את הקומפוננטה שלי? אני עבריר אליה את כל ה-props שהיא מקבלת (אני לא יודע אילו) וגם את התשובה לחיים עצם. בדיק כך:

```
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-`spread operator`:

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

אבל מה שהוא עושה זה לחת את כל ה-props שימושיים ולהעביר אותם, איך שם, בפורמט מסודר, אל הקומפוננטה שאינו עוטף, בלי שאני אctrיך להכניס אותם ידנית. נוסף על כן, אני מעביר גם את משמעות החיים.

זו דוגמה מאד תיאורטיבית, אבל היא מסיימת להבין עד כמה HOC הם פשוטים למטרות הסינטקס המאיים. פשוט לוקחים קומפוננטה אחת ומוסיפים לה תכונות או מתודות. הנה נראה דוגמה מעשית יותר. באחד הפרקים הקודמים יצרנו קומפוננטה שמנילה סטייט עם מספר, שאתחול ל-0, ולחיצה על הcptor גרמה למונה לעלות-1. מה הבעה? בכל פעם שביצענו רענון לדף, הסטט

התאפס והמונה חוזר ל-1. הסטייט נשאר בזיכרון אבל רק לאורך חיי הקומפוננטה. עם HOC אנו יכולים להעшир כל קומפוננטה שהיא ולחשוף API שיבצע שמירה של המספר הזה.

הקומפוננטה נראה כה:

```
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 ומוגנים להציב אותה אצלכם ולראות שהיא עובדת באופן מושלם. אך הבעיה היא שברגע שאתם טענים מחדש מחדש את הדף (באמצעות F5 למשל), המונה מתאפס וחוזר ל-0. אנו רוצים להעшир את הקומפוננטה זו באמצעות HOC על מנת לשמר את המונה בזיכרון הדף דן.

שמירה ב"זיכרון" של הדף, באופן ישיר או גמ רענון, נעשית באמצעות שמירה לאובייקט `localStorage`. מדובר בקובץ קטן שנשמר על המחשב ושומר נתונים גם לאחר רענון של הדף. הכתובת אליו מתבצעת באמצעות:

```
localStorage.setItem(key, value);
```

והקריאה מתבצעת באמצעות:

```
localStorage.getItem(key);
```

ראוי לציין שאין שום קשר בין `localStorage` לריект. מדובר בפייצ'ר של הדפסון ושל הג'אוوهסקריפט שרצ עליו, בדיק כמוה, `fetch`, למשל, המשמש ל-AJAX בסביבת שרת.

פונקציית ה-HOC שלי תיקח כל קומפוננטה ותוסיף לה מתודות `save` ו-`load`. לפונקציה אני אקרא `: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 דרךה שתי מתודות פשוטות: `save` ו-`load`.
מתודה `save` שומרת `localStorage` ומתודה `load` טוענת מה-`localStorage`.

ואיך אני מעביר קומפוננטה כזו? אני פשוט זכר שהקומפוננטה שעוברת דרך ה-HOC מקבלת ל-`props` שלה כמו 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;
```

בתוך קומפוננטת אב אני מבצע `import` לשתי הקומפוננטות שלי: ה-HOC והקומפוננטה שעוברת דרכו, ואז אני יוצר משתנה שאליו אני מחזיר את הקומפוננטה המועשרת:

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

בקומפוננטה זו אני יכול להשתמש כמו בכל קומפוננטה אחרת. צריך לזכור שה-HOC, במקרה זה, תמיד מחזיר לי קומפוננטה שנייה יכול להשתמש בה ב-JSON. זה בדוק מה שאני עושה בקומפוננטת הקונטינר:

```
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 :

```
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-handler increaseHandler שמו פועל לאחר לחיצה על הכפתור, אני קורא ל-props.save ומעבר דרכו שני פרמטרים: key שהחלמתי שהוא counter והערך, שהוא מספר.

שימוש ל-`useEffect`: על מנת למנוע שגואה אם הקומפוננטה לא עטופה ב-HOC, הוסף תנאי שבודק אם יש לנו אtribut `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 מתրנדורת לראשונה, אני אקרא למתודת `load` עם המפתח (שהוא `counter`), אמרו אותו במספר (כל הערכיהם ב-`localStorage` נשמרים כמחרוזת טקסט) ואכניס אותו לסטיט. 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 לבלבל. הנה ננתח אותו שוב. מדובר בפונקציה שמקבלת שני פרמטרים: פונקציה אונonymit וערך ריק. המערך הריק נועד להורות לריאקט להפעיל את הפונקציה האונonymit רק פעם אחת, בתחילת טיענת הקומפוננטה.

בפונקציה האונonymit יש קריאה ל-`props.load`, המתודה השנייה לנו על ידי ה-HOC. אם היא נמצאת אני מmir אותה במספר. אם לא, אני מחזיר 0 והכל נכנס לסתיט. זה הכל. אני בודק, כמובן, אם `props.load` קיימת.

از למה להכנס לכל הסרט הזה של ה-HOC ולא פשוט להכניס את ה-`localStorage.set` וה-`localStorage.get` לקומפוננטת `CounterUp` מההתחלה? למה לטrhoה לעטוף אותה ב-HOC? התשובה היא שעכשיו אני יכול לחת את ה-HOC הזה ולי עטוף אותו קומפוננטות נוספות ולתת להן, כבמטה קסם, את היכולת לקרוא ולכתוב לתוך ה-`localStorage` בלי להתאים כלל ובלוי לנכון שוב ושוב `localStorage.set` ו-`localStorage.get`. זה דבר חשוב מאד, כי לעיתים יש לנו פונקציונליות שאנו צריכים להטמע בקומפוננטה שלנו בנוגע לתקשורת לצד השרת וזה חוסך המון-המון קוד מיותר. הקריאה לצד השרת, או במקרה הזה ל-`localStorage`, תמיד יהיה מרכזota במקום אחד.

תרגיל:

הוסיף ל-HOC שכתבנו clear – Method שמנקה את ה-localStorage – WithStorage
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 props שבא לנו את הקומפוננטות שעוברות דרך ה-HOC. במקרה זה פשוט הוספנו מתודה נוספת `noscript` שנקראת `clear` ועושה את הפעולה הבאה:

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

תרגיל:

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

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

פתרונות:

```
import 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 להוסיף כפטור. הכפטור הוא כפטור ה-`:reset`

```

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

```

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

```

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

```

מאיופה ה-`props.clear` מגיע? מה-HOC שעוטף את הקומפוננטה. זו הסיבה שאנו בודקים אם הוא קיימ. איפה הוא עוטף את הקומפוננטה הזו? 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

לאוֹטִינְג



ראוטינג

עד כה עבדנו עם אפליקציות בננות דף אחד שבו יש קומפוננטות שונות. אבל זה כמובן לא ריאלי. באתר אמיתי, או באפליקציה אמיתי, יש כמה וכמה דפים. באתר אינטרנט בסגנון הישן, כל מעבר דף גרם לטעינה מחודשת של האתר כולו. ריאקט (וכן בכלל אפליקציות הווב המודרניות) מעבiri דפים נועשים ללא טעינה כלל מהשרת ובאמצעות קומפוננטות ריאוטינג שմבוצעות ניתוב של קומפוננטות דרך צד הקלוח, דבר המאפשר חוויה גלית טובה בהרבה. הטכנית זו נקראת SPA – ראשי תיבות של Single Page Application.

ראוטינג (Routing), או בעברית תקנית "ניתוב", הוא מונח ידוע בכלל הנוגע לאפליקציות וביבות, ומובן שיש אותו גם בריקט. ריאקט, בניגוד לפרימירוקים אחרים (כמו אングולר), לאקובעת איך למש את הריאוטינג, אלא יש כמה קומפוננטות ניוט חיצונית שמומלצות על ידי ריאקט וכל אחד יכול לבחור מה שהוא רוצה. אפשר גם למש את הריאוטינג בעצמכם, אם כי זה לא מומלץ בהתחשב בעובדה שיש קומפוננטות רבות אחרות שכבר נבדקו במאות ובאלפי אתרים אחרים. האובייקט `.history` המכיל את הכתובת נקרא `location`. אפשר למצוא אותו תחת `window` וכן יכולם לשנות אותו באמצעות `window.location`.

הבעיה היא ששינוי שלו גורם לטעינה חדשה זהה, כפי שכבר למדנו, גורם לרנדור חדש של כל הקומפוננטות ויוצר חוות משמש לא טובה, וגם עלול לגרום בעית ביצועים. הפטرون? לשנות את `location` בלי לטעון את הדף מחדש. יש שתי דרכים לעשות את זה: באמצעות שימוש באובייקט `History`, שנמצא תחת `window`, או באמצעות תוספת לכתובת בשם `hash`. השימוש ב-`window.history` נחשב למתקדם יותר והשימוש ב-`hash` נחשב לפשטוט יותר. הנה נדגים ריאוטינג באמצעות `hash` בלבד.

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

השרת אינו מסוגל "לראות" את מה שיש מאחורי הסימן `#` וכשאנו משלים אותו בדף, הוא לא נשלח כלל לשרת. זו הסיבה ששינוי שלו, אפילו ידני, לא יטען מחדש את הדף. בכלל זה פופולרי מאוד לראוטינג ב-`hash` כמו באפליקציות ריאקט.

על מנת לתרגל ריאקט נוצר שלוש קומפוננטות פשוטות שנעבור ביניהן. `Home`, `About` ו-`Help`. בדוגמאות כאן אני מכניס בהן רק כותרת פשוטה, אבל הן יכולות להכיל דפים שלמים וקומפוננטות אחרות, כמובן.

קומponentת `Home.jsx`:

```
import React from 'react';

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

export default Home;
```

קומponentת `About.jsx`:

```
import React from 'react';

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

export default About;
```

קומponeנטה Help.jsx

```
import React from 'react';

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

export default Help;
```

כאמור, אלו קומponeנטות פשוטות לשם לימוד ריאוטיניג והבנתו בלבד. כדי לבצע ריאוטיניג אנו יוצרים קומponeנטת אב שמכילה את סרגל הניווט ואת הקומponeנטה שתטען בכל פעם את קומponeנטת העמוד הפעיל, ככלمر את `About`, `Home`, או את `Help`.

קומponeנטה האב בעצם נתלית באירוע בשם `hashchange` שנמצא על `window`. האירוע הזה מתרחש בכל פעםISM שמשהו משנה את הכתובת שיש אחרי ה-#. כך, למשל, אם אני נמצא בדף `example.com#page1` – האירוע יופעל כתוצאה של קישור סמוביל אל `#page2`. על מנת לדעת מה הכתובת שאחרי ה-# אני משתמש בפקודה:

`window.location.hash.substr(1)`

הפקודה זו מורכבת משני חלקים – החלק הראשון:

`window.location.hash`

מחזיר לנו את כל מה שיש ב-#. במקרה זה `#page2`.

`substr(1)`

החלק השני מוריד את ה-# ומותירה אותנו עם `page2`.

نبין את זרימת המידע בקומפוננטה לפני הכתיבה שלה:

1. בתחילת הקומפוננטה מחברת לאירוע `hashchange` פעולה. הפעולה מופעלת בכל פעם שנלחץ קישור ל-#.
2. כשהפעולה מופעלת, אנו בוחנים את `window.location.hash` כדי לראות מה יש אחרי ה-#.
3. מכניםים לסטיט את ה-`route` הפעיל.
4. השינוי בסטייטגורם לקומפוננטה להתרנדר מחדש. ברגעור בודקים את ה-`route` הפעיל ובוחרים 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>
  );
}
```

```

        </div>
    );
}

export default Router;

```

את הקומponentה נציג כMOVEN ב-`App.js` באפליקציה Create React App 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 קורא בקומponent Router. הדבר הראשון שהוא עושים הוא ליצור סטייט בשם `route` ולתת לו ערך ראשוני של מה שיש בכתובת #:

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

למשל, אם אני טוען את הקומponentה כשהאני נמצא ב:

<http://localhost:3000/#/home>

הסטטוס יהיה `home`/. הקוד יחזיר לי את השורה הבאה:

```
useState(window.location.hash.substr(1))
```

הדבר השני שאני עושה הוא שאני גורם לקומפוננטה, באמצעות הhook `useState`, באנדרואיד הראשוני בלבד, להציג איזור שישנה את ה-`route` בסטייט בהתאם למזה שיש אחורי ה-#. אני מודאג שמדובר באנדרואיד הראשוני בלבד כי אני מעביר ב-`useEffect` פרמטר שני של מערכת ריק:

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

שימוש לב: מפני שהפרמטר השני שמועבר ל-`useEffect` הוא מערכת ריק [], אז הפונקציה שאנו מעבירים תופעל רק באנדרואיד הראשוני של הקומפוננטה – כלומר פעם אחת. זה חשוב, כי אם נציג מציגים איזור על `hashchange` בכל רנדור של הקומפוננטה זו תהיה לנו זיליגת זיכרון חמורה. אנו מציגים איזור פעם אחת בלבד.

הדבר השלישי הוא ליצור פונקציה פשוטה שיעשה בדיקה מה הסטייט הפעיל ומחליפה את משתנה Child בקומפוננטה המתאימה לסטטיו. שימוש לב שאני משתמש במשתנה שמתחליל באות גדולות – Child ולא child. זה משומם שמדובר במשתנה המכיל קומפוננטה ואני מצין בפנוי 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 import-ים נגעים מהצורה -About, -Help ו-Home.

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 מתרכזת ברכיב Child, שמשתנה בכל פעם. אני מציב את getChild לפני הקראיה לקומפוננטה עצמה. ככלומר המשמש לו חוץ על קישור, והאורווע של ה-hashchange עובד ומשנה את הסטייט. שינוי הסטייט גורם לרנדור מחדש, שבו Child מקבל קומפוננטה חדשה.

כאמור, כך זה נראה באופן מאד גס ופשוט. אם נרצה שזה יעבד עם קישורים אמיתיים ולא עם hash נדרש להשתמש ברכיבים שונים במקצת ולהפריד, כמובן, בין הקונפיגורציה לרכיב הראותיניג (למשל לשים את כל הנתיבים בקובץ JSON או על השרת), אבל בגודל – כך זה נראה. ראותיניג בריאקט אינו וודו וכל למש אותו, אבל בעולם האמיתי **נעדיף** להשתמש בקומפוננטה מוכנה מראש.

הקומפוננטה הפופולרית ביותר נקראת, בפשטות, react-router, ואני נלמד עליה בנוגע לראותיניג. אולם העקרונות המנחים אותנו רלוונטיים גם לראותיניג של כל קומפוננטה. הקומפוננטה הזו משתמשת ממש בכתובות אמיתיות ולא #-#, אך העיקרון זהה. react-router היא ספריית ראותיניג כללית וספרייה נוספת שלה, react-router-dom, היא המסייעת לאפליקציות ווב. אם כך, נלמד להשתמש בשתיهن.

ראשית, ניכנס לדף של react-router

<https://github.com/ReactTraining/react-router>

אפשר לראות שיש שם מדריך מסודר וdockumentציה טובה. זה כלל מפתח בוגע לכל קומפוננטה שאתם רוצים להשתמש בה בריאקט (ובכלל בכלל מערכת) – Dockumentציה טוביה היא הכרחית. אם אין Dockumentציה, עדיף לא להשתמש באוותה קומפוננטה. אבל זה לא המקרה פה.

כיוון שמדובר בקומפוננטה חייזנית, אנו נתקין אותה. ההתקנה נעשית, בדיק כmo כל קומפוננטה חייזנית אחרת, באמצעות `npm install react-router-dom` למדנו בקורס פרק על קומפוננטות חייזניות. נכנסים עם הטרמינל אל המיקום של הפרויקט שלנו ומתקינים את קומפוננטת react-router ואת router-dom באמצעות:

```
npm install react-router-dom
```

גם כאן, על מנת להציג, נשתמש בקומפוננטות `Home`, `About` ו-`Help`. בנגד דוגמה שהראינו, של ראוטינג בסיסי, `react-router-dom` עובדת עם כתובות אינטרנט אמיתיות – קלומר ללא הסימן `#`. יש לנו כמה יתרונות והミושן נעשה באוותה הדוחן.

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>
      </Switch>
    </div>
  )
}
```

```

        </Route>
        <Route path="/about">
            <About />
        </Route>
        <Route path="/help">
            <Help />
        </Route>
    </Switch>
</div>
</Router>
);
}

export default MyRouter;

```

אם עברתם על תת-הפרק שסבירו איך מבצעים את הרואוטינג ללא מודול חיצוני, זה יכול להיראות כמו פשטוט עד כדי גיחוך. מאחורי הקומפוננטה יש שתי תפיסות עיקריות:

1. כל קישור לנתיב נעשה באמצעות קומפוננטת `Link`.
2. הקומפוננטה הראשית היא רואוטר ו-`switch` ממומש לא ב-`switch` רגיל אלא בעזרת קומפוננטה.

זה הכל. ברגע שambilים איך רואוטינג עובד באופן טבעי, גם הקומפוננטות של הרואוטינג נראות פשוטות.

כל מואוד להשתמש גם בפרמטרים של URL. בדוקו מנטציה מסוימת אינן יכולה לעשות זאת. כל מה שעליינו לעשות הוא להוסיף את הפרמטר עם נקודותיים 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 מהרואוטינג הבסיסי יותר הוא:

<Route path="/help/:id">
זה בעצם מאפשר לנו להכנס פרמטרים, למשל:

<http://localhost:3000/help/6382020>

והפרמטר זהה יהיה זמין לנו בקומפוננטת `Help` באמצעות `useParams`, שהוא פונקציה שmaguaה 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. בדוקומנטציה של `react-router` יש דוגמאות רבות לשימוש שכדי לבחון. אבל כאמור, אם הבנתם איך ממשים ריאוטינג בעצמכם ואייך העיקרון עובד, לא יהיה לכם קשה להבין ולהשתמש בקומפוננטות של ניוט.

תרגיל:

צרו שני קומפוננטות: Home, שזיהה לקומפוננטה מתחילה הפרק, והקומפוננטה הזו, המציגת תמונה אקראיית מהרשף:

```
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.

פתרונות:

```
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`, שמוסע בכל פעם שה-`hash` (זה ה-`#`) ב-URL משתנה, מחליף את משתנה ה-`route` בהתאם לערך של מה שיש אחרי ה-`hash` וטוען קומפוננטה אחרת.

תרגיל:

ממשו את הפתרון הקודם באמצעות `.react-router`

פתרונות:

```
import 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>
        </Switch>
      </div>
    </Router>
  );
}
```

```

<Route path="/image">
  <NiceImage />
</Route>
</Switch>
</div>
</Router>
);
}

export default MyRouter;

```

כיוון שבמזהuber 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 שאנו צריכים.
4. עטיפת כל הקומפוננטה בקומפוננטה ראשית מסווג `Router`.
5. תפריט וקישורים באמצעות רכיב `:Link` `<Link to="/image">Nice Image</Link>`
6. יצירת קומponentת `Switch` במקום אמורות להיות הקומפוננטות שאנו מגיע אליהן בינויו.
7. להציב בתוך קומponentת `Switch` את הקומפוננטות שנטענו כתוצאה מפעולות הראוטינג.

למשל: `<Route path="/image"><NiceImage /></Route>`

תרגיל:

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.

הכניסו ניוט באמצעות פרמטרים, כך שתוכלו להכניס מספר ב-URL, למשל:

`http://localhost:3000/image/100`

או:

`http://localhost:3000/image/44`

ותמונה עם ה-`id` המתאים תיתען.

פתרונות:

קומפוננטת הרouting:

```

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>
  );
}

```

```

        </Route>
      </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;

```

.Route הוספת הפרמטר id ב-path בקומפוננטת – הפתרון הוא פשוט יותר – בראוטינג. בראוטינג שאמורה לטעות את id, הלווא היא Nicelimage, צריך לבצע import id-params. UseParams שמיינדה אלינו מ-react-router, מקבל את id וילש滔ל אותו ב-imgSrc. אפשר ממש גם שדה או תיבת Select שבהם כתבים מספר, והתמונה המתאימה תיתען.

פרק 20

הונתקה



קונטקסט

קונטקסט הוא מונח חשוב בראקט. הוא מאפשר לנו לחלק מידע בין קומפוננטות שונות, בדומה לסטיטיט, אבל הוא גלובלי. ביום אנו מכירים שתי דרכים להעברת מידע בקומפוננטות: האחת היא באמצעות `props`, כאשר קומפוננטה האב מעבירה לקומפוננטה הבנות מידע דרך ה-`props`, והשנייה היא אירועים – קומפוננטה הבת מפעילה אירוע כתוצאה מפעולה כלשהי (משתמש או טיימר). אם קומפוננטה האב העבירה פונקציה שmaps פעולה כתוצאה מאירוע כלשהו (`Handler`), הוא יופעל כאשר אירוע מופעל בקומפוננטה הבת.

הבעיה היא שזה יכול להסתבך. 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 שימוש. קומפוננטה מקבלת מצד השרת את המידע על המשמש וצריכה להעביר את המידע הזה הלאה. לעיתים יש מידע בקומפוננטה אב מסוימת וצריך להעביר אותו לקומפוננטה שנמצאת חמיש רמות מתחתיה בהיררכיה ולשרר את המידע. זה מורכב. קונטקסט מאפשר לנו לשמר מגיר מידע גדול שממנו כל קומפוננטה יכולה לקרוא, מעין סטייט גלובלי.

נכון לגרסה الأخيرة של ריאקט, מומלץ להשתמש בקונטקסט אך ורק למיידע שלא מתעדכן באופן תכוף. לעדכנים באופן תכוף מומלץ להשתמש ברידקס, שבו נדון באחד הפרקים הבאים, או באלטרנטיבות אחרות לניהול סטייט גלובלי כמו `mobx` או `observables`, שאין נלמדות בספר זה.

בקונטקסט יש שני משתתפים: האחד הוא הפרוביידר (`Provider`) והשני הוא הצרכן (`Consumer`). הפרוביידר חייב תמיד לעטוף את הצרכן ולהיות גבוה יותר ממנו בהיררכיה, אחרת הצרכן לא יוכל לגשת לקונטקסט שלו הפלביידר מספק. הפרוביידר יכול לאכלס את הקונטקסט באופן סטטי או לבצע קריאה ל-API חיצוני. הצרכן מקבל גישה לקונטקסט באמצעות הוקם.

פרופיביידר לא חייב להיות קומפוננטה ובודרך כלל לא בניו כקומפוננטה אלא כקובץ ג'אווהסקריפט רגול, למשל `UserContext.js`, שמנגדיר 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 לשימוש בפונקציית `Provider` יש לנו לפחות קומפוננטה אחת יוצרת קונטקסט באמצעות:

```
const UserContext = React.createContext({});
```

האוbjיקט הריק, שמועבר כפרמטר ל-`createContext`, הוא בעצם בריית מיחל – ערך שמוחזר רק כאשר קומפוננטה מבצעת קרייה לפרופיביידר זהה, אם הוא לא משוויך אליה. זה אידיאלי לבדיקות, אבל כרגע תשאירו אותו כאובייקט ריק.

אני משתמש ב-`API` של קונטקסט כדי ליצור וליצא את הפרוביידר, את הצריכה ואת הקונטקסט. אני יכול לבחור לפרופיביידר, לצריכה או לקונטקסט תחילה כרצוני. במקרה הזה, כיון שמדובר בקונטקסט של משתמש, בחרתי בתחילה של `User`.

אחרי שהגדרתי קונטקסט, אני יכול להשתמש בפروبיזידר. הפروبיזידר מספק לכל הקומפוננטות שutowפות אותו את הקונטקסט שהוא נותן. אני יכול לשימושו בכל מקום, אפילו ב-`App.js`. הינה 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 הפروبיזידר מהקונטקסט שיצרתי. כזכור, אני מיצא שם פروبיזידר, צרכן וקונטקסט.

```
import { UserProvider } from './UserContext'
```

אני בוחר את המידע שאני רוצה להעביר בكونטקסט. המידע יכול להיות סטטי או מידע שמניע משירות (כלומר מהשרת). בדוגמה זו אני משתמש במידע סטטי:

```
const user = {
  name: 'Ran',
  surName: 'Bar-Zik',
  city: 'Petah-Tiqwa'
};
```

הצעד האחרון הוא להשתמש בפروبיזדר כמו בקומפוננטה רגילה ולהעביר אליו את המידע. כל מה שיהיה עtopic בקומפוננטה זו וCompatibility כל הקומפוננטות שמתוחת לקומפוננטה העtopicה, כלומר אלו שתוחתיה בהיררכיה, יקבלו גישה לكونטקסט:

```
<UserProvider value={user}>
  Any component
</UserProvider>
```

עכשו במקום Any Component נשים קומפוננטה אמיתית, למשל Welcome.jsx

```
<UserProvider value={user}>
  <Welcome />
</UserProvider>
```

לא נשכח לעשות import, כמובן, לפני שאנו מציבים אותה.

איך הקומפוננטה הזו 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 הזו משתמשת בקונטקסט באמצעות הוק. השלב הראשון הוא להביא הוק, שעובד בדיקות כמו כל הוק אחר:

```
import React, { useContext } from 'react'

השלב השני הוא להביא את הקונטקסט שאנו מעוניינים להשתמש בו. במקרה זהה,
:UserContext

import UserContext from './UserContext'
```

השלב הבא והאחרון הוא פשוט לזכור את המידע שהפרוביידר הזה מביא לנו, באמצעות:

```
const user = useContext(UserContext);
```

מהשלב הזה, כל מה שיש ב-user הוא מה שיש לנו בפרוביידר.

אם הפורוביידר מתעדכן מסיבה מסוימת – הוא ירנדר מיד את הקומפוננטות שמשתמשות בו. שימושו לב שהקומפוננטה צריכה להיות מודעת ל-`UserContext` ולבסוף לו `import`.

אבל מה קורה אם אנו רוצים לעדכן את הקונטקסט ממוקם אחר שהוא לא הפורוביידר? אם אנו רוצים לחת לפורוביידר אפשרות לה汰עדן על ידי אחת מהקומפוננטות בתחום ההיררכיה, אנו צריכים להעביר, נוסף על הקונטקסט, גם פונקציה שתשנה אותו.

אני אדגים זאת באמצעות קומפוננטת `Container`. קומפוננטת `Container` ממשת פורוביידר שיש לו ערך שmagiu מהסתיט וגם פונקציה שיכולה לשנות את הסטיט:

```
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 שבמקום להכניס רק את ערך אובייקט המשמש, אני מכניס את ערך אובייקט המשמש בסטייט ומעביר אובייקט שיש בו גם את הסטייט וגם פונקציה שמשנה אותו.

אם Welcome.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.js`, ראשית אני שואב את אובייקט המשמש (כדי שאוכל להציג אותו) וכן את הפונקציה שמשנה את הסטייט של הפרוביידר. את הפונקציה אני מפעיל באירוע הלחיצה של הכפתור.

از בעצם מה שקרה הוא:

1. אני מגדר קונטקסט בקובץ נפרד.
2. בוחר היכן להציב את הפרוביידר שלי, בדרך כלל בקומפוננטת אב. במקום שבו אני מציב את הפרוביידר, אני מזריק ערך. הדרך הכח טובה לעשות זאת היא באמצעות סטייט, אם כי זו לא חובה.
3. אני יכול להגיד ייחד עם הערך פונקציה שמשנה את הסטייט של הפרוביידר. כל מי שמשתמש בكونטקסט זהה יכול (אם הוא רוצה) להשתמש בה ולשנות את הקונטקסט לכלם.

كونטקסט הוא דרך מצוינת לחלק מידע בין קומפוננטות שונות וגם קל להשתמש בו עם הוקים. הדרך לעדכן קונטקסט יכולה להיות מסורבלת, אבל אם מבינים שברגע שיוצרים קונטקסט ומידע גם יוצרים פונקציה שמשנה אותו ומעבירים את שנייהם – אז הכל נהייה פשוט.

תרגיל:

העתיקו את קומפוננטת Welcome.jsx וקומפוננטת Container.jsx ל-Create React App Container.js. שימו לב שה-.js.app מכיל את ה-.jsx ב敞开 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;
```

הוסיפו כקומפוננטת בת לקונטינר קומפוננטה בשם CityName.jsx שמציגה את שם העיר של המשתמש.

פתרונות:

ראשית, ניצור את הקומפוננטה. הקומפוננטה היא קומפוננטה שאמורה לצורך קונטקסט ונראית כלא:

```
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 מבחינתנו הן:

```
import React, { useContext } from 'react'
const user = useContext(UserContext).data;
```

כעת ניבא את הhook `:useContext`

כיוון שאנו יודעים שהקונטקסט מגיע כאובייקט שתכונה אחת שלו היא המידע ותכונה נוספת היא הפונקציה שמעדכנת אותו, אנו מעוניינים רק במידע ואנו מכניסים את ה-`data` אל המשתנה `user`. מהה מדבר בהדפסה פשוטה.

ומה קורה בקומפוננטת `Kontainer`? אין חדש. אנחנו רק צריכים להציב את קומפוננטת `CityName.jsx` מתחת לפורוביידר:

```
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 הוא בת, נצדה, נינה או הבית של הנינה של `UserProvider` זה לא משנה – היא יכולה להשתמש בكونטקסט.

פרק 21

חיבור לשורת נס סרווייסים



חיבור לשרת עם שירותיים

פרק זה מניח ידוע מוקדם ב-AJAX ובבדל בין הבקשות השונות – לפחות ברמת פונקציית ה-`fetch` הטבעית.

חיבור לצד שרת הוא קרייטי בכל הנוגע לאתר או לאפליקציה המבוססים על ריאקט (או בכלל). רוב האתרים אינם אתרים סטטיים אלא אתרים המתעדכנים במידע המגיע מהשרת. המידע המגיע מהשרת נכנס כבר כ-`props` לקומפוננטות מקומפוננטות אב כלשהן או לكونטקסט. את הקראיות אלו מבצעים באמצעות AJAX ואפשר להשתמש ב-`fetch` שmagiu עם ג'אווהסקריפט או בכלל ספרייה אחרת, כמו הספרייה הנפוצה `Axios`.

כשאנו מרכיבים קוד בריект, הוא רץ בדף בלבד. נכון, Create React App מרים שרת פיתוח במחשב שלנו לשם הנוחות. אבל בסופו של יום, בסביבת הפודקשן, הסביבה האמיתית, השרת יכול להיות שונה לחלווטין. הדף יוכל לקבל אתקובצי הג'אווהסקריפט שמורכבים מספריית ריאקט ומהקוד שלכם מכל שרת שהוא. אחרי שדף הלקו מוריד אתקובצי הג'אווהסקריפט האל, הוא מרים את האפליקציה הבנויה על ידי ריאקט על הדף בלבד. אם הוא צריך מידע מהשרת, הקוד שלכם אמרור להביא לו אותו.

בדרך כלל, מתכווני צד השירות כתובים API בצד השירות כדי שייחזר לכם מידע, והאחריות של מתכווני ריאקט היא לשולח בקשות לשרת. כן, למשל, אתם משתמשים בראוטינג דף התחברות עם שם משתמש וסיסמה. המשתמש מקליך שם משתמש וסיסמה ולוחץ על כפתור "שלח". האחריות שלכם היא לשולח, באמצעות AJAX, את שם המשתמש והסיסמה ל-API של צד השירות. צד השירות אומר להחזיר לכם תשובה של 200 ועובדיקט המכיל את פרטי המשתמש, בהנחה שהcoil מצלי, ולשתול עוגייה בצד המשתמש. האחריות שלכם היא לחת את המידע הזה ולהכניס אותו לكونטקסט (למשל) או לרידקס, שנלמד בהמשך, כדי שכל המידע באתר יוצג למשתמש המחבר כמו שצರיך.

כמו בראוטינג, ריאקט אינה קובעת עבור המשתמש איך לשולח את הבקשות ואין לנול אותן. אפשר לבצע את הבקשות בעזרת ה-`fetch` הטבעי שיש בג'אווהסקריפט בסביבת הדף או בעזרת ספרייה. הספרייה הפופולרית לניהול בקשות AJAX היא `Axios`. אנו נתרגל באמצעות `fetch`.

יש כמה דרכים לנצל את הבקשות – אפשר לנתח אותן בתוך הקומפוננטה ממש. אנו נתאמן עם ה-API של בדיחות צ'אק נוריס, שבעצם מה אפשר לנו לקבל בבדיקה צ'אק נוריס בקישור הבא:

<https://api.chucknorris.io/jokes/random>

נסו בעצמכם! העתיקו והדביקו את הקישור בדף וצפו ב-JSON שmagiu.
איך נממש את זה? כמובן – אפשר בקומפוננטה:

```
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 קריאת ה-AJAX, באמצעות `fetch`, אל פונקציה מסודרת שאנו מפעילים רק פעמיות, באמצעות ההוק `useEffect` שמקבל כפרמטר ראשון את הפונקציה להפעלה וכפרמטר שני מערך ריק, שבעצם אומר שהוא יופעל רק כאשר הקומפוננטה מורנדרת. הפונקציה שעושה גם מכינה את הבדיקה המתבקשת אל הסטייט.

הקוד הזה, שאמור להיות מובן מאוד אם אתם כבר מכירים AJAX וידעוים איך `fetch` עובד, הוא קוד ולידי, אבל הוא רעיון גrosso מאד אם האפליקציה שלכם מעט יותר מורכבת, למשל אם יש דרישות נוספות כמו להכניס את תוכאות הבדיקה לקונטנסט, כדי שעוד קומפוננטות יוכל להציג אותה, או לשנות את ה-API. זו הסיבה שמקובל מאד להכניס קריאות לשרת לפונקציות ג'אויסקייפט טהורות שנקראות שירותים (Services). יוצרים בפרויקט תיקיה שנקראת `Services` ושם שמים את כל הפונקציות המטפלות בקריאות לשרתים.

למשל, אם אני רוצה להמיר את הקריאה לקומפוננטה זו בקריאה לשירות אמיתי, השירות יראה כך:

```
export default function getChukJoke() {
  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 לשום לב שלא משנה איזו דרך נבחרה – מדובר בג'אויסקייפט טהור. ריאקט לא מערבת בעניינו השירותים וכל אחד יכול למשר אוטם לפי הבנתו.

סביר להניח שכאשר תעבדו מול מערכות אמיתיות, מי שיגדר לכם איך השירותים מתנהלים מול השרת יהיו מתכנתיה הבק אנד, והם אלו שיגדרו לכם איך להתחבר.

תרגיל:

נדרשתם להציג בדיחת אבא רנדומלית באתר. מתכונת צד השרת הגדר לכם API באופן הבא: יש לשלוח בבקשת GET אל הכתובת:
<https://icanhazdadjoke.com>

עם בקשה ה-GET דואגים לשלוח את ה-`header:Accept: application/json` כדי לקבל את בדיחת האבא.

רמז: בבקשת fetch עושים באופן הבא:

```
return fetch(`https://icanhazdadjoke.com` , {
  headers: {
    'Accept': 'application/json'
  },
})
```

פתרונות:

```
export default function getDadJoke() {
  return fetch(`https://icanhazdadjoke.com` , {
    headers: {
      'Accept': 'application/json'
    },
  })
    .then((response) => {
      return response.json();
    })
    .then((jsonObject) => {
      return jsonObject.joke;
    });
}
```

משתמשים ב-service זהה כה:

```
import 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;
```

בגدول, זה לא ריאקט אלא ג'אוوهסקריפט. הסרוייס הוא פונקציה שמייצרת בקשהAJAX. הבקשת נשלחת עם header ומחזירה אובייקט JSON. אנו לוקחים את האובייקט ומחזירים את ה-joke מהתוכן.

הריект פשוט קורא לפונקציה שמחזירה פרומיס וכאשר הוא מתמלא, אנו מכניסים את מה שהוא נותן לנו לסטיביט של joke.

בשלב זה אנו יכולים לטפל, כמוון, בשגיאות או בפלט לא צפוי. למשל, משהו בסגנון זהה:

```
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!'); }
      );
  }

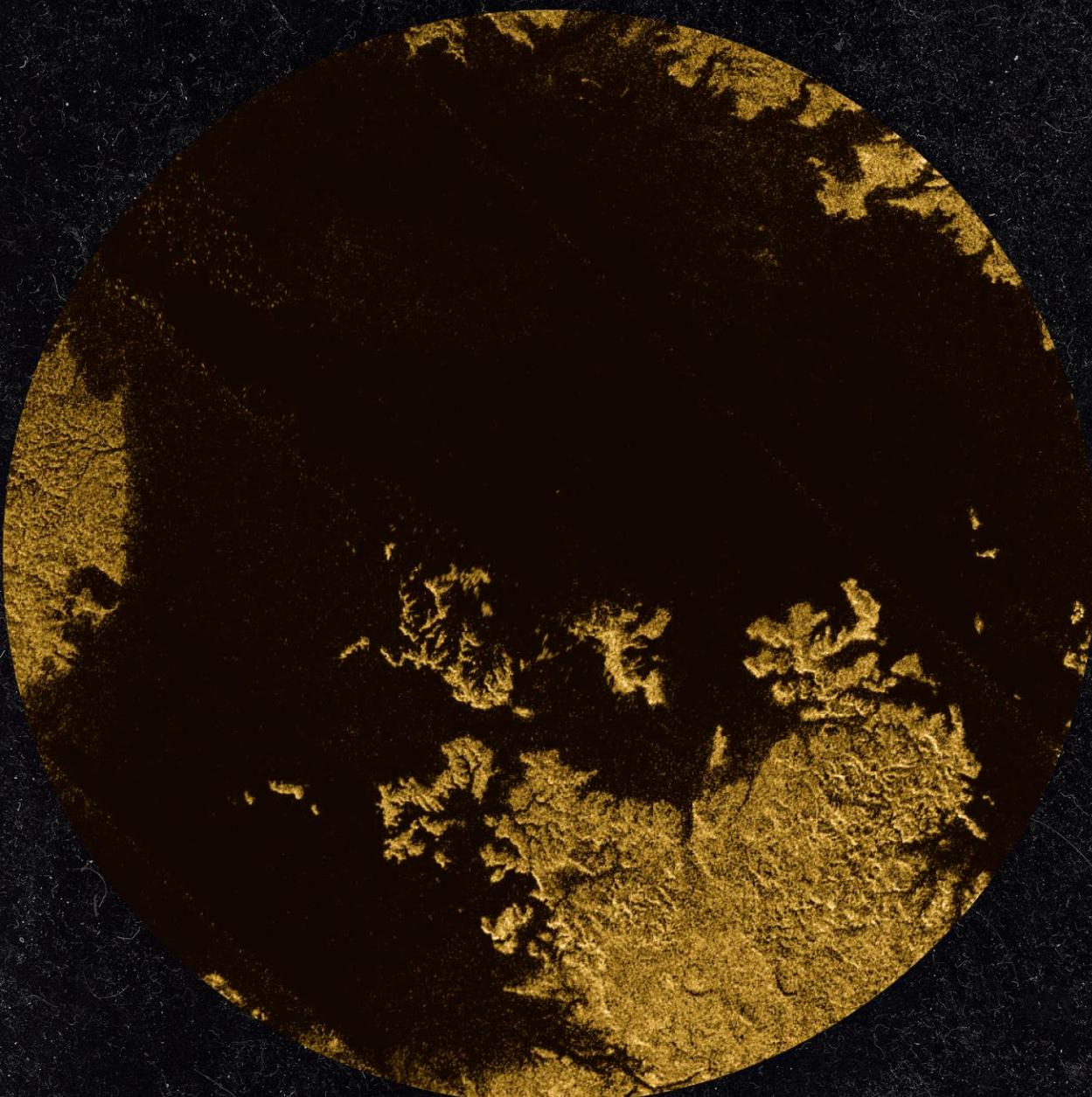
  return (
    <div>
      {joke}
    </div>
  );
}

export default DadJoke;
```

או כמוון ניהול מסודר יותר. חשוב מאד, בקריאה AJAX, לחשב על מה שקרה אם השירות לא מחזיר את התשובה המczופה.

פרק 22

מבוא לבדיקות נס



מבוא לבדיקות עם jest

בדיקות אוטומטיות הן חלק חשוב מפיתוח מקצועי בפרויקט. בפיתוח יש לנו כמה סוגי בדיקות: בדיקות ייחודית (הבודקות את הקומפוננטה), בדיקות פונקציונליות (כדי לראות איך הקומפוננטה עובדת), בדיקות אינטגרציה (הבודקות איך הקומפוננטות השונות מתחשרות עם צד השרת) ובבדיקות End to End (E2E) שבודקות את כל האפליקציה מקצה למקצה.

בדיקות חיוניות לפיתוח נכון, וברוב החברות הטובות נהוג כתוב בבדיקות ייחודית. הבדיקות הללו קריטיות כיון שם מתכונת אחר יעשה שינוי בקומפוננטה שלכם, הבדיקה תישבר והוא יבין שיש בעיה עם תסריט מסוים שבניתם עליו ובדקתם אותו. זו הסיבה שבבדיקות הן דבר חשוב ולא מיועד רק לאנשי QA. בעולם הפיתוח מקובל שמפתחים כתובים בבדיקות ייחודית.

בדיקות אוטומטיות הן סקריפטים, הכתובים בג'אוויסקריפט, ויש לכל בדיקה קритריונים להצלחה. למשל: "אם לווחצים על כפתור, האפליקציה קוראת לשירות". הבדיקות הללו רצות באופן נפרד באמצעות פקודה מסוימת וMRIIZIM אונן לפני כל הכנסת קוד חדש. בכלל זה בדיקות אוטומטיות הן חלק מהה Shnkeria Continuous Integration, שהוא תהליך שקוד חדש עבר כשהוא משולב בקוד שנכתב קודם לכן. אם מישחו כתוב תוספת לקוד שלנו (למשל הוסיף props נוספים) ויש לנו בבדיקות טבות שרצות על הקוד החדש וכל הבדיקות רצות אף אחת מהן לא נכשלת – אנו יודעים שכנהראה נכתב קוד איקוני (או לפחות צזה שעבר את הבדיקות) ושוב נוכל להעלות אותו לאויר.

ספריית התוכנה שMRIIZIM את הבדיקות בפרויקט נקראת `jest`. היא כבר מגיעה עם Create React App באופן מובנה. משתמשים ב-`jest` במקומות נוספים מעט ריאקט והוא ספרייה פופולרית מאוד לבדיקות. בבדיקות מקובל שקובץ הבדיקות נושא את השם של הקובץ המקורי עם התוספה `spec` או `.test`. למשל, קובץ הבדיקות של `Welcome.js` יהיה `Welcome.spec.js`, אבל יש Kongnaciot נוספות.

הבה נכתוב בדיקה אוטומטית ראשונה עבור `xsj.Welcome`, שתוראה 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, להדפיס `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 לrendsר את הקומפוננטה. אנו נבדוק את תוכנת `innerHTML` של הקומפוננטה כדי לראות שהיא הדפסה את הפלט כמו שצריך. כך עושים את 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 מיבאים את ריאקט (היא נדרשת על מנת שנוכל לрендר קומפוננטות) ואת react-dom, וכמובן 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 פונקציה Hz שבתוכה הבדיקה.

חלק שלישי – הרצת הקומפוננטה

הצעד הראשון הוא ליצור HTML שבתוכו הקומפוננטה מתrndr. מדובר בג'אוסקריפט 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, שלא עשינו import, והשני הוא המיקום שבו היא מתrndרת, במקרה זה ה-DIV שיצרנו.

חלק רביעי – הבדיקה

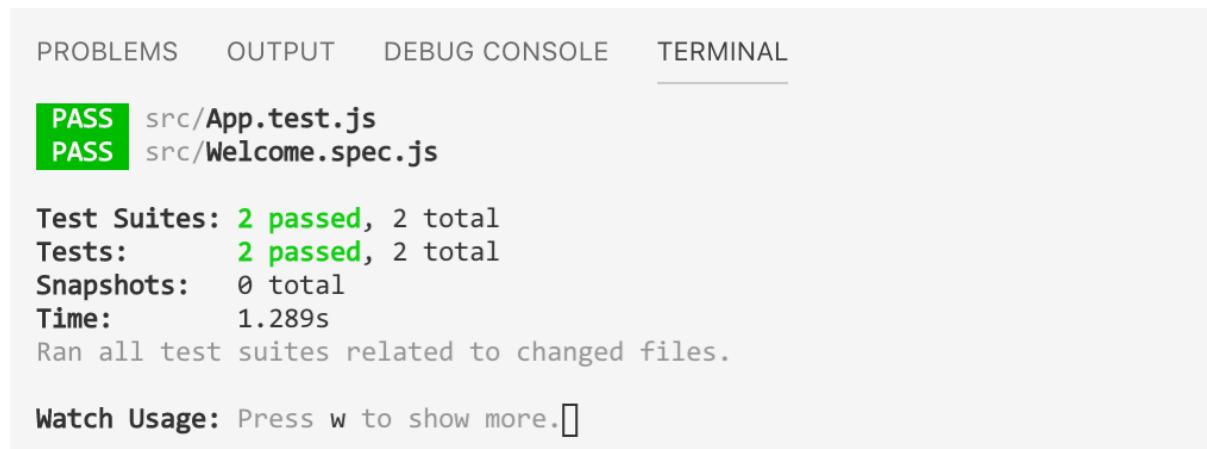
אחרי שהקומפוננטה רונדרה, אנו יכולים לבדוק את ה-DIV שבתוכו היא נמצאת. אנו עושים את הבדיקה באמצעות פונקציית expect. מדובר בפונקציה שגמ היא באה עם jest והוא מקבלת ביטוי שאליו יש לשדר את ההנחה. במקרה זה הביטוי הוא div.innerHTML === div.innerHTML – כלומר "הביטוי שווה ל...". כל בדיקה היא בעצם ביטוי והנחה. ההנחה היא מתודה שאליה אנו צריכים להעביר את הטקסט שאנו משווים אליו.

האמת היא שבמקרה זהה ההסבר הרבה יותר מסורבל מהקוד:

```
expect(div.innerHTML).toEqual('<p>Hello world!</p>');
;
```

הקוד הזה מדבר בעד עצמו והוא מובן מאוד. אחרי שכתבנו את הבדיקה הגיע הזמן לראות שהיא עוברת בהצלחה. את הבדיקות אנו מרכיבים בטרמינל שכבר למדנו לעבוד איתו בפרקם קודמים. אם אנו עובדים עם Visual Studio Code נפתח חלונית של טרמינל באמצעות בחירה בלשונית Terminal בחלק העליון של התוכנה ואז ב>New Terminal

לחולופין, 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 התהליך ואם הבדיקה תעבור בהצלחה, תראו אישור.



The screenshot shows the Visual Studio Code interface with the 'TERMINAL' tab selected. At the top, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. Below the tabs, the terminal window displays the following output:

```
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 למשל טעיתי בכתיבת הבדיקה ובמקום לבדוק שהקומפוננטה מדפסה Hello world ב-w קטעה, הנחותי שהיא מדפסה Hello World ב-W גדולה. רואים את השוני.

בדיקות לקומפוננטה עם props

אם לקומפוננטה יש props, علينا להעביר אותם בבדיקה. זה קל יותר ממה שזה נשמע. הנה נבחן את הקומפוננטה זו:

```
import React from 'react';

function Welcome(props) {

  return (
    <p>Hello, {props.name} {props.surName}!</p>
  );
}

export default Welcome;
```

כאן נעשה שימוש ב-props. כיצד אבדוק את הקומפוננטה? ראשית, אצור את קובץ הבדיקות. כמעט דבר לא משתנה מלבד הרנדור של הקומפוננטה. מנ הסתם אני ארצה לרנדר אותה עם ה-props המתאיםים. איך עושים את זה? כך:

```
import 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>');
});
```

השוני העיקרי הוא השורה זו:

```
ReactDOM.render(<Welcome name="Moshe" surName="Cohen" />, div);
```

אם אני בוחן props, אני נדרש להעביר אותם ולבוחן את התוצאה.

ברוב הפעמים נשמש בספריות עזר לשם הבדיקות, כמו ספריית העזר Enzyme שמסייעת מאוד בבדיקה אירופית וברנדור של הקומפוננטות והקומפוננטות הבנות (היא לא נלמדת במסגרת הספר).

פונקציית בדיקה – **assert**

בבדיקה שעשינו לשם הדוגמה, השתמשנו ב-`toEqual`. כמובן, משווים ממש את הפלט. הקוד שבו השתמשנו הוא:

```
expect(div.innerHTML).toEqual('<p>Hello, Moshe Cohen!</p>');
```

הוא מורכב מכמה חלקים. החלק הראשון הוא פונקציית `expect`, שמגיעה עם jest ומקבלת ערך כלשהו, טקסטואלי או לא. החלק השני הוא `toEqual`, שמבצעת את ההשוואה. ההשוואה הזאת נקראת פונקציית `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 שהtekst 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, העיקרי שהוא מדבר 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 שימומשית אם רוצים לדעת שבמקרה זהה הבדיקה מחרירה לנו מהו שלא קיים. למשל, אם אני רוצה לוודא שאין `p` לקומפוננטה – כיוון שכמה קומפוננטות עם אותו `p` יכולות לעורר בעיות ב-DOM – אני אבצע את הבדיקה באופן זהה:

```
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');
```

תרגיל:

נתונה הקומפוננטה Greeting. כתבו לה בדיקה שבודקת שהוא מדפסה את הפלט Good Morning Moshe! כשם עביריים לה 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>');
});
```

בדיקת כמו בדוגמה שהובאה בפרק, ייבאנו את הקומפוננטה ובאמצעות `ReactDOM.render` ייצרו אותה בתוך ה-`div`. הבדיקה האמיתית היא השוואת הפלט של הקומפוננטה באמצעות `div.innerHTML`. את הרצה אנו מבצעים באמצעות `npm run test`.

הפלט ייראה 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.

```

תרגיל:

שנו את הבדיקה ובדקו רק שהטקסט Good Morning נמצא שם, ללא הפרמטרים.

פתרונות:

```

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`, כיון שהיא שמעניין אותנו הוא המילה הזו ולא כל הבדיקה. בדרך כלל נשתמש ב-`toContain`, כיון שאם נשנה את המבנה הפנימי של הקומponentה לא נשבר את הבדיקה.

תרגיל:

נתונה הקומפוננטה `asj.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:

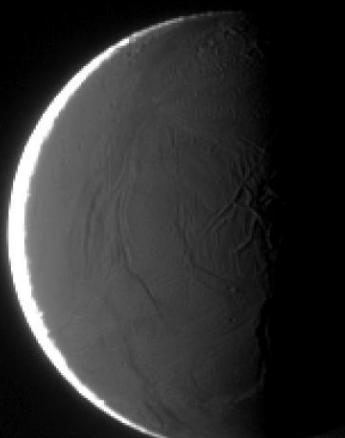
```
import React from 'react';
import ReactDOM from 'react-dom';
import Random from './Random';

it('Contains Good Morning', () => {
  const div = document.createElement('div');
  ReactDOM.render(<Random />, div);
  expect(div.innerHTML).toBeTruthy();
});
```

כתיבת הבדיקה הזאת היא פשוטה, כיוון שמדובר בקומפוננטה מאוד לא מסובכת. אבל כשאנו רוצים לבדוק את הפלט שלו אנו בבעיה, כי הפלט משתנה בכל פעם. אנו לא יכולים להשתמש בפונקציית ההשוואה `ToContain` או `toEqual` כי הקומפוננטה מייצרת בכל פעם מספר רנדומלי. מה הפתרון? להשתמש ב-`toBeTruthy` כדי לוודא שיש משהו ב-`innerHTML`.

פרק 23

יצירת בילד והעלאת האפליקציה לסביבה חיה



יצירת בילד והעלאת האפליקציה לסביבה חיה

אחרי שכתבנו את אפליקציית הריאקט שלנו – כתבנו קומפוננטות משלנו, השתמשנו בקומפוננטות חיצונית, כתבנו שירותים ופונקציות – הגיעו העת להעלות אותה אל השרת. צריך לזכור שבסוף כל יום כל האפליקציה היא קובץ ג'אווהסקריפט, קובץ HTML וקובץ CSS שנטענים על ידי הדף. בקובץ הג'אווהסקריפט זהה, הדחוס והמחובר, יש הנול: ספריית ריאקט, babel, הקומפוננטות החיצונית והקוד שנכתבם. הכל בקובץ אחד. הדף טוען אותו והכל מתחילה היבנות ממש. יש כאלה שבוחרים לחלק את הקובץ הדחוס לכמה חלקים – למשל תלויות חיצונית (ספריות צד שלישי כמו למשל UI React Material שעלייה למದנו באחד הפרקים הקודמים) והאפליקציה שכתבנו. אבל בסופו של הבילד, התוצאה היא קבצים דחוסים.

הקובץ הזה אמור להגיע לדף איכשהו. כשאנו בסביבת הפיתוח, עם Create React App, האפליקציה יוצרת לנו שרת קטן על המחשב (מבוסס Node.js), יוצרת לו כתובת (localhost:3000) ואיפילו פותחת את הדף ומן אותה שטוען בתורו את קובץ הג'אווהסקריפט. אבל מה קורה באתר אמיתי?

באתר אמיתי אנו צריכים שרת אמיתי שיגיש את הקבצים לדף. כשהאני אומר "יגיש" אני מתכוון לתהילך שבמסגרתו אנו מקלדים כתובת של אתר אינטרנט בדף. הדף ניגש לשרת האתר והוא מחזיר לו קבצים שהדף מציג.

כך למשל כשהאני גולש לגוגל, אני מקליד בדף google.com, השרת של גוגל מעביר לדף דף HTML, קובץ ג'אווהסקריפט ואת הלוגו של גוגל (או כל דודל אחר של גוגל שיש שם במקום הלוגו). בסופו של דבר יש שרת שמעביר לדף קבצים.
אילו קבצים?

1. דף HTML שיש בו קישור אל:
2. קובץ CSS של עיצוב.
3. קובצי הג'אווהסקריפט המכוצעים של האפליקציה.

איך אנו יוצרים את השרת זהה? יש הרבה דרכים. הדרך פשוטה ביותר הוא לשכור shared hosting באתר כלשהו ופשוט להציב עליו את קובץ ה-index.html וקובצי ה-CSS והג'אוסקריפט (זה עובד אם עושים רואוטינג מבוסס #), באמצעות העתקה פשוטה, בתיקייה הראשית של השרת בעזרת הממשק של ניהול השרת שספק השירות מעניק לכם. הקולדת כתובה שם המתחם וכניסה אל האתר תגרום לטעינת קובץ ה-HTML ומכובן כל הקבצים המוקשרים אליו.

יש דרכים רבות אחרות – לא חסרות שפوتצד שרת שיש בהן שירותים. הפופולרית ביותר היא Node.js, אך יש גם את ג'אווה, C# ופייתון שמאפשרות יצירת שירותיים. אם אתם, או מתכנתים לצד השרת בחברה שלכם, יוצרים שרת כזה – בסופו של דבר תצטרכו לגורם לו להחזיר ללקוח את דף ה-HTML והקבצים. הלקוח ייכנס, הקבצים ייטענו ומפה התפקיד האפליקציה לפעילה בדיקון כמו Create React App.

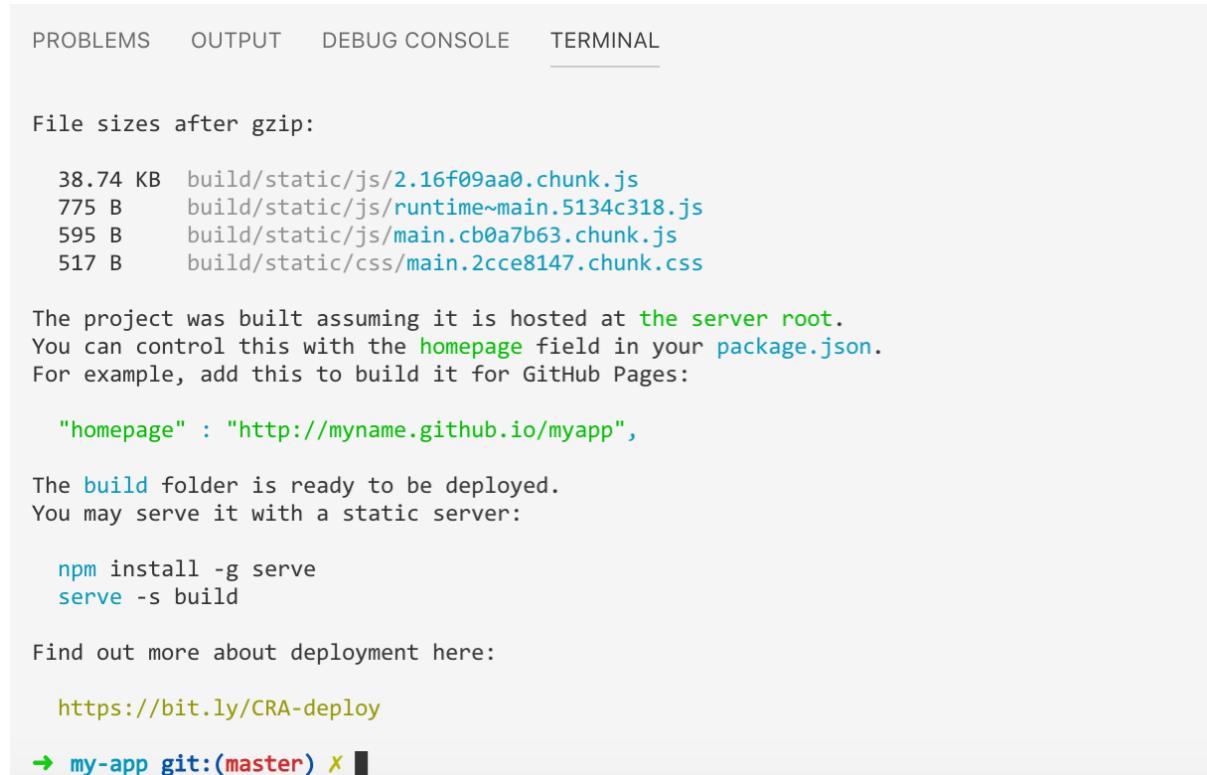
איך מייצרים את קובץ ה-HTML, קובץ CSS ומכובן קובצי הג'אוסקריפט? באמצעות פקודת wkhtml מינימלית שיש להקיש בטרמינל – ממשק הפוקודה הטקסטואלי. אם אתם משתמשים ב-Visual Studio Code, לחזו על Terminal בתפריט העליון ואז על New Terminal. יפתח לכם חלון בתחום הממשק עם שורת פוקודה טקסטואלית.

אם אין לכם Visual Studio Code, היכנסו אל ה-cmd בחלונות ונווטו אל התיקייה של האפליקציה.

מיד כשאתם יכולים להקליד בטרמינל, הקלידו:

```
npm run build
```

ירוץ תהליך שמאחד ומכווץ את קובצי ה-`js` ו-`CSS` ויצרת קובץ `index.html` וגם קבצים סטטיים נוספים שימושיים לשרת. כל הקבצים האלה ייכנסו אל תיקיית `public` – זו התקינה שיש להעלות אל השרת כדי להשתמש באפליקציה שלכם, בין שמדובר בשרת של חברה גדולה ובין שמדובר ב-`shared hosting`:



```

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 בילד והוא נפוץ באפליקציות צד לקוח כמו ריאקט, אングולר ודומותיהן. התהליך הזה 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 הקלדת הפקודה זו בטרמינל:

```
npm install -g serve
```

ולאחר מכן:

```
serve -s build
```

וכניסה אל:

`localhost:5000`

ההבדל בין השיטה זו להקלהת `start` מוקה הוא שכאן אנו מרים את הבילד ממש, ככלומר את הקבצים המוגמרים שאנו אמורים להעלות לשרת. מודול `serve` יוצר לנו שרת ומעלה את התקינה `build` כדי שנוכל לבדוק את האפליקציה שלנו, מתיקות `-p 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`

תרגיל:

קחו כמה קומפוננטות שיצרנו בתרגילים הקודמים ושלבו אותן באפליקציית הריאקט שיש ב-[Create React App](#). הריצו את הבילד ובדקו אותו.

פתרונות:

יש להריץ `npm run build` ולה Смотрות לתוכה. אחרי שהתהליך מושלם, תיווצר אצלכם תיקיית `build` עם קובץ HTML,קובצי תמונה, CSS והכי חשוב: ג'אווהסקרייפט.

על מנת לבדוק את הבילד, אפשר להשתמש ב-serve מקומי כפי שלמדנו. Tipatch لكم בדף אפליקציה מלאה שנמצאת ב-localhost. אפשר להעלות אותה לכל שרת באמצעות העתקת תיקיית `build` והעברתה לשרת מבוסס Apache,Node.js או כל שרת אחר.

פרק 24

ריזוף



רידקס

חלק ממשמעותי 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 המונח קומפוננטות לצרכים לקלוט אובייקט עם פרטיו המשמש, קומפוננטה אחת תציג ברכה, "שלום רן", קומפוננטה אחרת תציג קישור למודול (פוף-אפ) פרטיו משתמש שבו יש קומפוננטה שאמורה להכיל את אובייקט המשמש. קומפוננטה נוספת, בתחתית הדף, צריכה לקלוט גישה לאובייקט כדי לדעת אם להציג פרטים משפטיים מסוימים ללקוח, וכך הלאה. יש הרבה מאוד קומפוננטות לצרכים גישה לפרטים האלו.

כדי שהשמחה תהיה כפולה – עולה השאלה מה קורה אם הסטייט מתעדכן. למשל אם המשמש שינה את שם המשמש שלו? או, למשל, אם יש מונה הודעות – מה קורה אם הקורא קרא הודעה אחת ויש להוריד את מונה ההודעות?

זו הסיבה שאנו זקוקים לסטטייט משותף של האפליקציה, סטייט שאפשר לעדכן מכל קומפוננטה וסטייט שאפשר לקרוא מכל אפליקציה. מצד שני, חשוב לשמר על הסדר, כדי שלא כל שינוי בסטייט המשותף יrndר את כל הקומפוננטות, גם כאלה שלא משתמשות בחלק מהסתטייט הששתנה; וכן שמי שירצה לשנות את הסטייט יוכל לעשות זאת, אבל רק את החלק שלו.

בפרק על קונטיקסט הסברנו על סטייט משותף והראנו איך עושים זאת זה עם קונטיקסט ועם הוקים. ללא ספק זו הדרך שリアקט מתקדמת אליה, אבל נכון לגרסה הנוכחית, קונטיקסט הוא לא תחליף סטייט משותף מכמה בחינות, במיוחד מבחינת ביצועים: קונטיקסט לא נועד לשינויים תכופיים אלא להעברת מידע סטטי (או מידע ששואף להיות סטטי) כמו טמפליט, אובייקט משתמש וכו'. לעומתו, ספריית רידקס (Redux), ספרייה חיצונית שעלייה נלמד בפרק זה, מוכוונת לדברים שימושיים כל הזמן. נוסף על כך, לרידקס יש שני פיצ'רים שימושיים מאוד: מסע בזמן, שמאפשר לנו, באמצעות כל היכולות של רידקס, לזרז לאורך חי האפליקציה ולראות מה השתנה ולשנות אותו בפועל, ו-middlewares – שכבת תוכנה נוספת בין הנטונים לאפליקציה.

כיוון管理 סטייט עומד פעמים רבות בפני עצמו ואיינו קשור בהכרח לריאקט, יש כמה ספריות לניהול סטייט משותף – הראשית והחשובה שבהן היא רידקס. אפשר להשתמש ברידקס גם

בפרימורקים אחרים והוא חיצונית לריקט, אבל בשנים האחרונות היא הפכה למזהה מאוד עם ריאקט.

רידקס נחשבת למרכיב משתי סיבות עיקריות: הראשונה היא הקונספט – הבנת הקונספט של מה רידקס עשו ואיך היא מנהלת סטייט מושתף. השנייה היא הקוד – תוספת הקוד של רידקס היא שימושית ומסובבלת. כדי שקומפוננטה תוכל להשתמש ברידקס יש לעתוף אותה בקוד רידksi זהה מורכב. מעבר לסרבול הקוד, רידקס נוטה להဏפה ולהכיל גם סטייטים שלא אמורים להיות גלובליים.

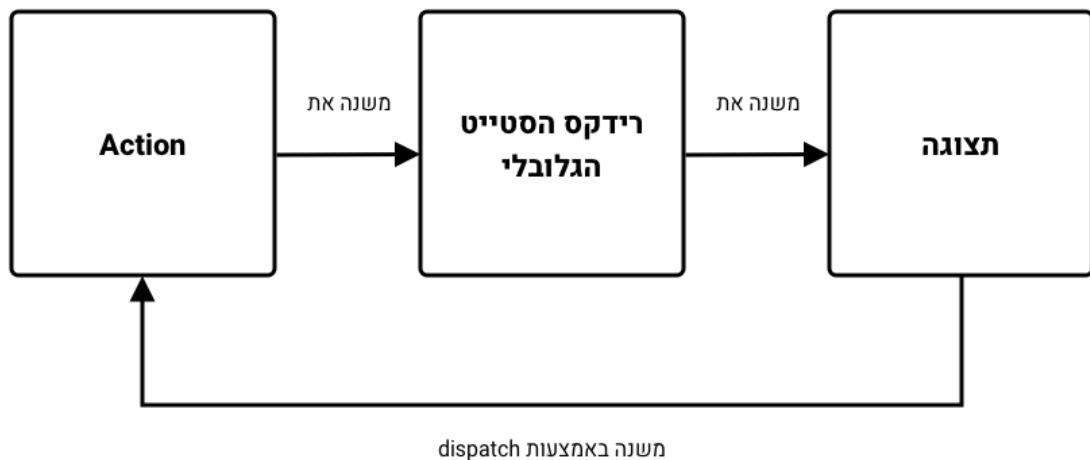
לכן אני ממליץ להשתמש בקונטקסט באפליקציות קטנות יותר ובירקס באפליקציות גדולות יותר, במקום שתוספת הקוד והמורכבות שווה יותר.

הפילוסופיה מאחורי רידקס

רידקס דוגלת בשלושה עקרונות עיקריים:

1. מקור אחד של אמת (Single Source of Truth) – יש סטייט אחד ויחיד והוא הנקודה שבה יש את המידע האמתי והאמין ביותר.
2. הסטייט הוא קרייה בלבד – אי-אפשר לשנות את הסטייט באופן ישיר. אפשר רק לקרוא ממנו.
3. שינויים נעשים באמצעות פונקציות טהורות – אם אנו רוצים לשנות את הסטייט, אנו עושים זאת באמצעות פונקציות שאין להן השפעות צדדיות ובהינתן אותו קלט, עושות בדיקות אותו הדבר.

אם ננסה להמחיש את זה באמצעות תרשימים, נראה שיש לנו סוג של מעגל עם כיוון אחד של זרימת מידע. אנחנו מתחילה בפעולה, שמופעלת על ידי קומפוננטה כלשהי באמצעות dispatch. הפעולה זו משנה את הסטייט הגלובלי. אף אחד אחר, חוץ מאותו action, לא יכול לשנות את הסטייט. הסטייט בתורו משנה את התצוגה ואז המשמש יכול שוב לעשות פעולה שימושית את המעגל.



כרגע זה נשמע מאד אבסטרקטי, אבל כשהנלמד איך מיישמים רידקס בפועל, נראה איך העקרונות האלה מתחממשים.

קומפוננטות ניהול מול ניהול

כשאנו מסתכלים על קומפוננטות בריאקט, הן מחולקות לשני סוגים:

קומפוננטות תצוגה או קומפוננטות ניהול – קומפוננטות שלא מודעות לרידקס ולא משתמשות בו. קומפוננטות אחרות מנהלות אותן באמצעות `props` ומקבלות מהן מידע באמצעות אירועים. הן מגדרות איך מידע נראה.

קומפוננטות ניהול – קומפוננטות שמודעות לרידקס ועוטפות על ידו. הן מנהלות קומפוננטות אחרות ומעבירות אליהן את המידע שמניע מרידקס. הן מקבלות מידע מרידקס על ידי רישום לאירועי רידקס ומשנות מידע על ידי `dispatch` (הפעלה) של פועלות (`actions`) רידקסיות. הן מגדרות איך מידע מתנהל.

קומפוננטות ניהול**קומפוננטות ניהול**

מתמקדות באין דברים עובדים

מתמקדות באין דברים נראים

מודעות לרידקם

לא מודעות לרידקם

נרשומות לסתיט הרידקסי

קוראות למידע מה-props

משתמשות ב-`dispatch` על מנת להעביר פלט
לרידקס

משתמשות בקולבקים כדי להעביר פלט

נקודות על ידי רכיב רידקסי עוטף

נקודות ביד

כשאנו מתכנים את מבנה הקומפוננטות שלנו, אנו מבצעים את הפרדה בין הקומפוננטות המנהלות והקומפוננטות המנהלות, אלו שנן עטופות על ידי רידקס.

בדוגמה שלנו נדגים באמצעות אפליקציה של רשימת קניות. הינה התוכנון המקורי:

תפקיד	שם קובץ	מה היא אמורה לעשות
אפליקציה ריאקט	App.js	מכילה את קומפוננטת הקונטינר.
קונטינר	Container.jsx	מכילה קומפוננטה של <code>ListInput.jsx</code> שבה אפשר להכנס מידע לרישימה.
קומפוננטה שלא מודעת לרידקס.		מכילה קומפוננטה של <code>ListView.jsx</code> שבה הרשימה מוצגת.
קומפוננטה המתקבלת קלט	ListInput.jsx	אלמנט <code><input></code> פשוט שבו משתמש מקליד את פרטי הפריט להכנסה לרישימה.
קומפוננטה המציגת פלט	ListView.jsx	קומפוננטה המכניתה מידע לרידקס.
		הציגת הרשימה. קומפוננטה מקבלת מידע מרידקס.

אחרי שמייפינו את הקומפוננטות השונות ואני יודעים אילו קומפוננטות יש לנו, זה הזמן לבנות את האפליקציה ללא רידקס.

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 כמודול בשורש הקומפוננטות. כל תפקידה של האפליקציה הוא להביא קומפוננטת `Container` וולדנו עליה בעבר. קומפוננטת `Container.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 רידקס, הקומפוננטה זו הייתה מורכבת מעט יותר, כיוון שהיא הייתה צריכה לנהל סטייט שה-`ListInput` היה מעודכן וה-`ListView` היה מקבל ומציג, או לעטוף את הקומפוננטות

האלו בكونטקסט פרוביידר ולגרום להן לעשות את זה. כיוון שיש לנו RIDKS, אלו 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 הקומפוננטה שבה יש שדה טקסט וכפתור (שניהם מוגעים במספרית קומפוננטות חיצונית) וסטיטט פנימי משלה. כאשר לוחצים על הכפתור, מה שיש בסטייט `item` אמור להגיע לסטיטט **globally** של RIDKS.

המטרה שלנו: להכניס את סטייט `item` לרידקס.

קומponeנטה ListView.jsx

```
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;
```

הקומponeנטה זו מציגה מערך של פרוטוים כרשימה. היא לוקחת את props.items (אם הוא לא קיים, היא מצביה מערך ריק) וממירה אותו לרשימה באמצעות הקומponeנטות החיצונית List ו-.ListItemText

המטרה שלנו: שהקומponeנטה תקבל את props.items מרידקס.
עכשו אנו יכולים לגשת להטמעת RIDKS.

הטמעת רידקס

התקנת רידקס

רידקס היא ספרייה חיצונית ועלינו להתקין אותה באמצעות npm, שלמדנו עליו בפרק על התקנת קומפוננטות חיצונית. ההתקנה היא פשוטה ומתבצעת באמצעות הקלה הפקודה הבאה בטרמינל, במקום של הפרויקט שלנו:

```
npm install react-redux
npm install redux
```

תכנון הסטייט וזרימת המידע במערכת

הסטיטיוט הגלובלי שלנו הוא בעצם אובייקט ג'אווסקריפט פשוט שיש בו מערך שיכול, תיאורטית, להכיל רשימה של דברים. למשל:

```
{
  listItems: [],
}
```

אחד העקרונות המרכזיים של רידקס הואuai-אפשר לגעת בסטייט ישירות. אז איך אני משנה אותו? באמצעות action. למשל:

```
{ type: 'ADD_ITEM', payload: 'Item #1' }
```

הסוג הוא סוג שאני מגדיר, במקרה זהה הוספה של פריט לרשימה. ה-payload הוא תוכנה שאני בוחר. זה יכול להיות payload או itemText או כל תוכנה אחרת שאני בוחר. אני מחייב, אך מקובל להשתמש ב-member שנקרא payload.

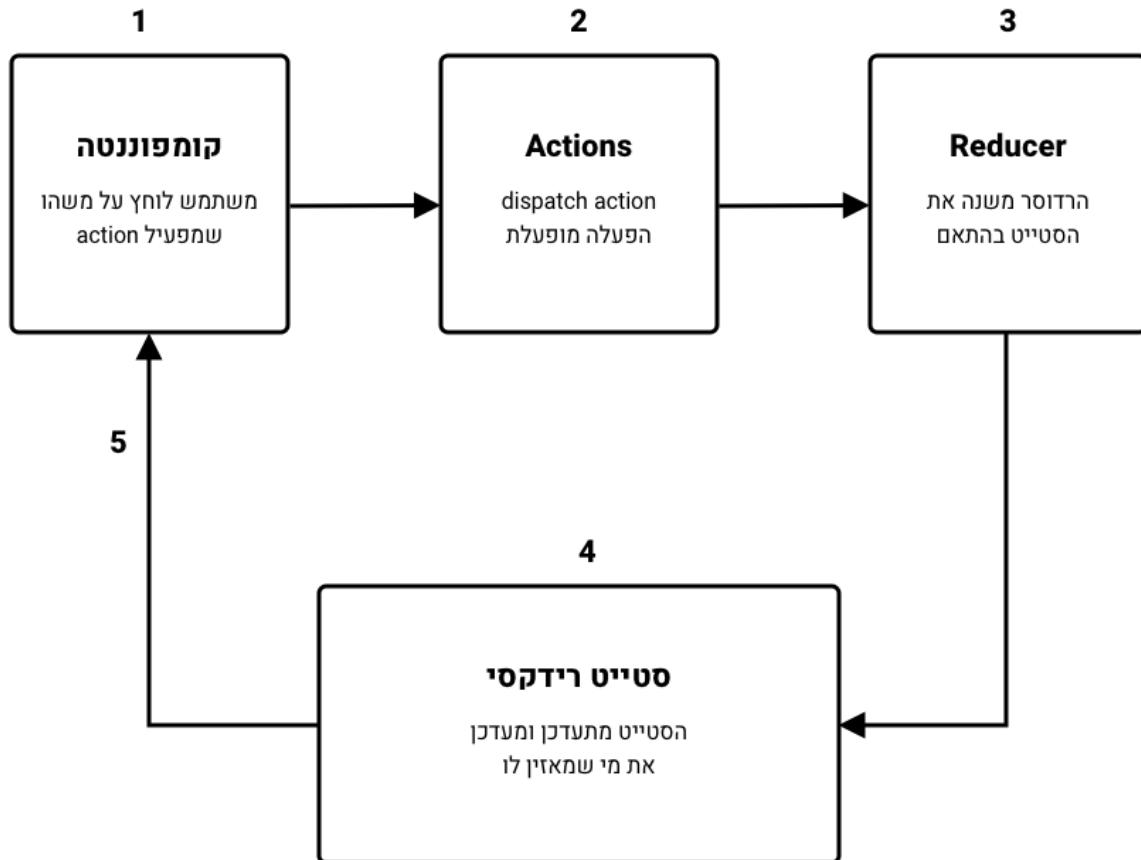
אחרי שהחלמתי על ה-action, אני צריך ליצור פונקציה שתיקח אותו ותכניס אותו אל הסטייט. חשוב מאוד להחזיר אובייקט חדש לסטיטייט כדי להראות לרידקס שצריך לנדר הכל מחדש.

משתמשים בטריק פשוט כדי ליצור אובייקט חדש, במקרה זה:

```
function listReducers(state = [], action) {
  switch (action.type) {
    case 'ADD_ITEM':
      return { list: [...state, action.payload] };
    default:
      return state;
  }
}
```

לפונקציה זו יש שם מפחד: `reducer`. התפקיד שלה הוא לנקח action ולהחליט איך הוא משפיע על הסטייט הגלובלי. במקרה זה, אם ה-`action` הוא מסוג `ADD_ITEM`, ה-`reducer` לוקח את ה-`action.payload` ומכניס אותו אל תוך הסטייט הגלובלי, אל חלק ה-`items`. פונקציות ה-`reducer` הן פונקציות טהורות (pure functions), פונקציות ג'אווהסקייפיות שאין מושנות את הקלט ומחזירות פלט קבוע על אותו קלט. כדאי לשים לב שחייב להיות `default` שמחזיר את הסטייט המקורי.

از הינה ה-`useState` הבסיסי: כשהאנו רוצים להוסיף פריט לרישימה, אנו יורим את ה-`action`, ה-`reducer` וופס אותו ומעדכן את הסטייט הגלובלי.



זה יכול להוישם מבלבל, אבל זה לא מאד מסובך. בגדול, זה מה שהוא:

mapStateToProps

בסוף דבר, גם כשאנו משתמשים ברידקס, אנחנו כותבים קומפוננטות ריאקטיות.

1. קומפוננטה ריאקטית מקבלת `data` מבוחר באמצעות `props`.
2. אנחנו רוצים להעביר לקומפוננטה שלנו נתונים מהסטור.
3. אז בעצם אנחנו רוצים לmaps חלקיים מהסטור לתוך `props` של הקומפוננטה שלנו.

mapDispatchToProps

הקומפוננטה שלנו מודיעת על שינויים באמצעות פונקציות שהיא מקבלת ב-props. אנחנו רוצים לגרום לשינוי בסטור.

1. שינוי בסטור נעשה באמצעות dispatch ל-actions.
2. אז אנחנו רוצים להעביר לקומפוננטה שלנו פונקציה שכש被执行ים אותה היא יורה dispatch עם ה-action הנכון.

השימוש של רידקס במערכת שלנו

חשוב לציין, לפניה שמתחלים, שמדובר בתהליך ארוך ולא פשוט. זו הסיבה שההמלצה היא ליישם רידקס רק באפליקציות מורכבות. על אפליקציה קטנה עם כמה עשרה 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 './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
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, { list: { items: [] } },
  enhancers);
```

```
ReactDOM.render(<Provider store={store}><App /></Provider>,
document.getElementById('root'));

serviceWorker.unregister();
```

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 הפטחים 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 הסטייט הרידקסי הראשון, שנקרא store. אפשר להתייחס אליו כל בסיס נתונים גלובלי שמחזיק את כל הסטייט ולעתוף אליו את ה-App שלנו:

```
const store = createStore(rootReducer, {
  list: [{}],
}, enhancers);

ReactDOM.render(<Provider store={store}><App /></Provider>,
document.getElementById('root'));
```

הערה חשובה: אל תבלבלו מהשם הזהה. הפרוביידר כאן הוא של רידקס ואינו קשור לكونטקסט.

כעת אנו מוכנים לעבוד עם RIDKS.
השלב השני הוא ליצור את ה-actions, הפעולות שבמצאים להן dispatch. אנו ניצור תיוקית redux בפרויקט שלנו ותת-תיוקיה בשם actions. שם ניצור קובץ `index.js` 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 ייחודי.

הfonקציה הזו היא פונקציה (בשם action creator) שקומפוננטות יכולות להשתמש בה כדי לשחרר action, שאותו ה-reducer תופס. ה-action נראה כך:

```
{ type: 'ADD_ITEM', payload: 'Item #1' }
```

ובן שהטקסט משתנה בהתאם למה שנרצה. במקרה זה, מי שיפעל את הפונקציה יוכל להעיבר שם ארגומנט כרצונו.

מה שיש פה עוד – אף על פי שפונקציונלית זה לא נדרש – הוא export לשם פעולה שהוא באוטיות גדולות. מי שישתמש בשם action הוא גם ה-reducer. מקובל לעשות את זה כדי למנוע בעיות של שגיאות כתיב.

ה-reducer תופס את ה-action וצריך ליצור אותו. ניצור תיקייה reducer ושם ניצור קובץ בשם listReducers.js. התוכן שלו יהיה:

```
import {
  ADD_ITEM,
} from '../actions'

function manageList(state = { items: [] }, action) {
  switch (action.type) {
    case ADD_ITEM:
      return { list: [...state, action.payload] };
    default:
      return state
  }
}

export default manageList;
```

ה-reducer הזה יעבד בכל פעם שנוצר action. אם יש action עם השם ADD_ITEM, הוא מכניס את ה-text שהוא איתו אל מערך items, יוצר אובייקט חדש ומחזיר אותו. כבר אמרנו קודם שיצירת האובייקט החדש היא קריטית לאופטימיזציה של RIDKS.

כדי שRIDKS ידע על reducer זהה, אנו צריכים להוסיף אותו לעולם. באותה תיקייה של reducer ניצור קובץ index.js ובו נכניס את התוכן הבא:

```
import { combineReducers } from 'redux'
import list from './listReducers'

export default combineReducers({
  list,
})
```

אנו משתמשים כאן בפקודה שנקראת `combineReducers` כדי לשינך את ה-`reducer` שלנו לחלק בסטיוט הקשור אליו. במקרה זהה החלטתי שהחלק של הרשימה בסטיוט הגלובלי ייקרא `list`, כלומר שהסטיוט הגלובלי ייראה כך:

```
state.list.item = ['item #1', 'item #2', 'item #3'];
```

זהו, עד כאן ה-`setup`, וזה עבודה קשה. אבל מהרגע הזה אפשר להשתמש ברידקס בקומפוננטות שיש להן נגיעה לסטיוט. קומפוננטה שצרכיה לשנות את הסטיוט תקבל, דרך ה-`props` שלה, את ה-`action` שמשנה את הסטיוט, שאותו כבר כתבנו.

קומפוננטה שצרכיה לקבל מידע מהסטיוט תקבל, דרך ה-`props` שלה, את החלק בסטיוט שהוא נדרש לקבל, וגם רידקס תרנדר מחדש את הקומפוננטה הזו בכל פעם שהסטיוט הספציפי הזה ישתנה.

נתחיל בקומפוננטה שצרכיה לשנות את הסטיוט. אני ארצה להעביר אל ה-`props` את ה-`action` המתאים: `addList`. איך אני עושים את זה? באמצעות `connector` שמחבר את הקומפוננטה הזו לRIDKS ומעבירות אל ה-`props` שלו, את ה-`action` שהוא נדרש.

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);
  }

  return (
    <div>
      <input type="text" value={item} onChange={changeHandler}>
      <button onClick={()=>props.addToList(item)}>Add</button>
    </div>
  );
}

export default connect(null, mapDispatchToProps)(ListInput);
```

```

};

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 `xsj.js` לkomponenta RIDKSIOT:

צעד ראשון: ייבוא

```

import { addItem } from './redux/actions';
import { connect } from 'react-redux';

```

לייבא את החלקים של RIDKSIOT (טוב, זה קל). אחד מהם הוא ה-`action` וחלק נוסף הוא פונקציית החיבור, שנקראת באופן מפתיע `.connect`.

צעד שני: למפות בין ה-`action` שאנו צריך ל-`props`

```

const mapDispatchToProps = (dispatch) => ({
  addToList: (item) => dispatch(addItem(item))
})

```

הfonקציה החשובה ביותר – אני בעצם יוצר פונקציה שמקבלת ארגומנט dispatch ומחזירה אובייקט. האובייקט נראה כך:

```
{
  addToList: function(item) {
    return dispatch(addItem(item));
  }
}
```

כלומר, הפונקציה זו דואגת לשים את addToList ב-props של הקומפוננטה. אם מישהו יפעיל את props.addToList, האירוע item הוא קריטי כי הוא בעצם הtekסט שמווער.

הfonקציה זו היא אחת המבלבלות ברידקס, אבל צריך לזכור שהוא בעצם הקשר בין ה-action ל-.props

צעד שלישי: להפעיל את ה-action שיש לנו ב-props איפה שצריין

```
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, הלווא הוא item, משוגר אל ה-action ומשם אל הסטייט הגלובלי. ה-action נמצא ב-props והוא לשם על ידי פונקציית החיבור.

צעד רביעי: ליצא את הקומפוננטה באמצעות connect

```
export default connect(
  null,
  mapDispatchToProps
)(ListInput)
```

על מנת שרידקס ידע על כל מה שעשינו, אנו לא מיצאים את הקומפוננטה באמצעות `export default` כרגיל, אלא עטופה במתודת `connect`. הארגומנט הראשון שומר לפונקציית מיפוי בין ה-`props` לסטיט. במקרה זהה הקומפוננטה לא משתמשת בסטייט הגלובלי (רק משנה אותו, לא צורכת ממנו) ולפיכך הוא יהיה `null`. הארגומנט השני הוא המיפוי בין ה-`actions` לפרופס שלו. הסברנו בסעיף הקודם. לבסוף – אנו מעבירים את הקומפוננטה עצמה. מהנקודה זו לא צריך לעשות כלום. כל מה שתכניiso בשדה הקלט יוכנס לסטיט הגלובלי. אתם יכולים להכניס `console.log` כדי לראות שהכל עובד כמו שצריך.

עכשו נשים אל הקומפוננטה שמציגה את הסטייט, הלווא היא `ListView.jsx`. כך היא נראה:

```
import React from 'react';
import { List, ListItemText } from '@material-ui/core';
import { connect } from 'react-redux';

const mapStateToProps = (state) => {
  return { ...state, items: state.list.items || [] }
}

function ListView(props) {
  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 צריכים ל-`connect`.

צעד שני: מיפוי

```
const mapStateToProps = (state) => {
  return { ...state, items: state.list.items || [] }
}
```

נראה מפיח, אבל מה שחשיבותו הוא החלק הזה:

```
items: state.list.items
```

אני בעצם אומר שמה שבסטיטו `props.items` הוא הסטיטו הגלובלי-<`list`->`items` – זו פונקציית המיפוי.

צעד שלישי: ליצא את הקומפוננטה באמצעות `connect`

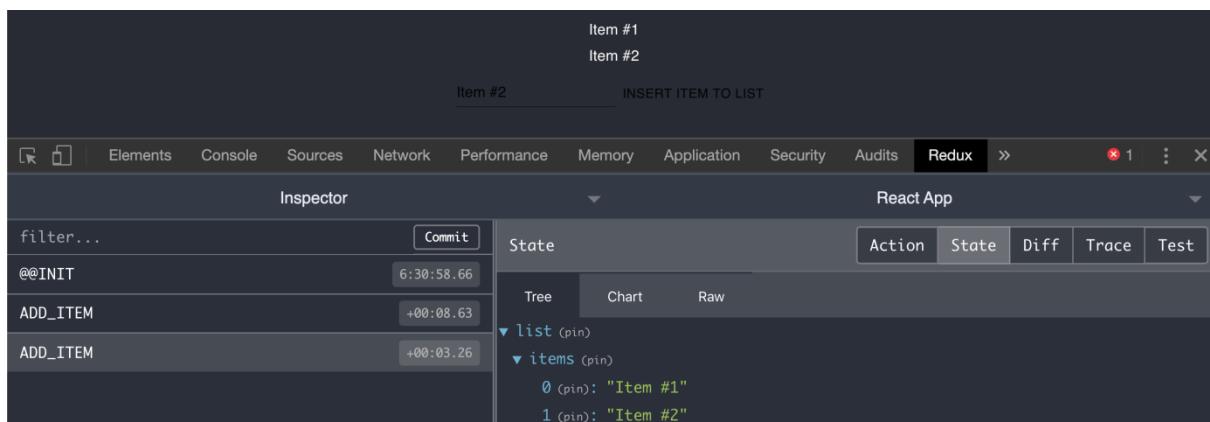
```
export default connect(
  mapStateToProps,
  null
)(ListView);
```

כמו בקומפוננטה שכתבה לסטיטו, גם הקומפוננטה זו צריכה להיות מיוצאת באמצעות `connect` שמקבל ארבעה ארגומנטים. הראשון הוא המיפוי בין ה-`props` לסטיטו הגלובלי. השני הוא המיפוי בין ה-`props` ל-`action`. כיוון שהקומפוננטה היא רק קוראת ולא כתבת לסטיטו הגלובלי, אני אעביר שם `null`. וכמוהן, `ListView` הוא הקומפוננטה.

אגב, ניתן בהחלט מצב שבו יש קומפוננטה שגמ קוראת מהסטיטוּט הגלובלי וגם כותבת לו, ואז אנו נעביר בפונקציית `connect` ששתי פונקציות מיפוי ולא יהיה שם `mapDispatchToProps`.

אחרי שתעשו את כל זה, תוכלו לראות שהאפליקציה ציה באמצעות עובדת ואפשר להוסיף להוסף לה איבטים בראשימה. זו פעולה מטאורפית לאפליקציה כל כך קטנה, אך אפליקציות מורכבות יותר מאוד יהנו מRIDKS.

כדי לנצל במקרה אחוז את היכולות של RIDKS, כדי להתקין את כל הפתחים של RIDKS, שזמן גם בפירופוקס וגם בכרום. יש לחפש Redux DevTools בchnoot התוספים ולהוריד אותו בחינם ובמהירות. לאחר ההתקנה תראו לשוניית RIDKS בכל הפתחים. לחיצה עליה תראה לכם את הדיבאגר של RIDKS. בדיבאגר אפשר לבחון את כל מה שקרה בסטייט הגלובלי ולבוחן היבט כל `.action-1 action-2`



אפשר לנوع קדימה ואחוריה בזמן ולראות מה השתנה. ממשך השימוש של כל הפתחים פשוט ונדיי לנסה אותו.

RIDKS יש הרבה יותר מניהול סטייט. יש לנו גם MiddleWares, שיכולים ליצור עוד תהליכי, נוספים על שינוי הסטייט, ולסייע לנו בניהול ובתפעול מתקדם של האתר ושל האפליקציה. לימוד RIDKS הוא מטלה קשה ואפיילו מסובכת, ויצא לה (בצדק או שלא הצדק) שם של מהهو שקשה לעשות, אך בהחלט שווה להתאמץ כיון שנכון להיום – כל אפליקציה גדולה ואתר גדול משתמשים בכל ניהול סטייטים, RIDKS היא אחד הכלים הפופולריים.

תרגיל:

הוסיפו כפטור איפוס רשימה כרכיב נפרד ששמו הוא `ResetButton` וימוקם ב-`אקס.js`.

פתרון:

ראשית ניצור את ה-`action`. `ה-action` יגרום לרשימה להתאפס. בקובץ `jsx.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 הקיים 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:
      return { list: [...state, action.payload] };
    case RESET_LIST:
      return { item: [...state, []] };
    default:
      return state
  }
}

export default manageList;
```

הקוד פשוט הופך את state-items לאובייקט ריק ומשכפל אותו להלאה. להזכירכם, אלו חיבורים להחזר אובייקט חדש כיוון שכן המנווע של RIDKES ידע לレンדר את הקומפוננטה. הייצור של האובייקט חדש נעשה כך:

```
return { item: [...state, []] };
```

הצעד השלישי הוא ליצור את קומפוננטת Button ולהחבר אותה לרידקס ולפעולת RESET_LIST שהפונקציה resetList יורה.

նיצור פונקציה :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 לתוכה `props` והפעלה שלו תפעיל את `dispatch(resetList)`, שזו הפעולה. ומובן לחבר את `mapDispatchToProps` באמצעות `:connect`:

```
export default connect(
  null,
  mapDispatchToProps
)(ResetButton);
```

הפעולה الأخيرة שיש לעשות היא להוסיף את רכיב `ResetButton.js` אל ה-`Container.js`.

```
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 קsha להוסיף את RIDKS לאפליקציה שלנו, הוספה של פעולות הופכת לפחות מואוד.

פרק 25

סיום - ומה העכשווי?



סיום – ומה עכשווי?

אם קראתם את כל פרקי הספר, אתם יודעים לבנות אפליקציות ריאקטיות. אבל הידע שלכם הוא עדין תיאורטי, ולימוד פרימומורק חדש נעשה באמצעות הידיים. אי-אפשר למדוד פרימומורק אם לא עברתם את השלבים שאף ספר לא יכול להבהיר אתכם: התסכול המתלווה למשהו שלא עובד כשרה, השמחה הגדולה כאשר מצליחים לפטור את התקלה ורוכשים ידע נוסף יקר מפז, החיפוש המתמיד בגוגל וב-stackoverflow ובנויות מוצריים אמיתיים ופתרון התקלות שיש בהם.

אם סיימתם לקרוא את הספר ולפטור את התרגילים, הצעד הבא שלכם הוא לבנות אתר אפליקציה או יישום מבוססי ריאקט. זה הזמן להציגך לקבוצה שבונה אתרים או אפליקציות בהתקנות עבור מיזמים הקרובים ליביכם, לבנות אתר או אפליקציה לחבר שתמיד היה לו רעיון או לעמומה שתמיד רציתם לתרום לה. הניסיון המעשוי הראשון של בנית אתרים ובכתבת קוד יהיה עבור עמותת טולקין (מחבר "שר הטבעות") בישראל – אין סיבה שהזה לא יהיה כך עבורכם. הניסיון כאן הוא קריטי ואתם חייבים לבנות משהו משל עצמכם, אפילו ברמת הניסיון. אסור להסתפק בתרגילים שיש בספר.

כאשר אתם מחליטים לבנות משהו משלכם – מילת המפתח היא תכנון. אחד ממנהלי הפיתוח שלי נהג לומר ששבועיים של תכננות חוסכים שעתיים של תכנון. חשבו על הקומפוננטות, בדקנו באילו קומפוננטות חיצניות אפשר להשתמש ואיך הן עובדות ואם אפשר להשתמש בהן עבור האתר שלכם. בנוסף לכך, בקשו לקבל ביקורת על העבודה שלכם וכך תגלו איך תוכלו יכולות לעשות אותה טוב יותר.

אבל אחרי שבניתם את הפרויקט שלכם – יש עוד כמה צעדים שתצטרכו לעשות כדי להשתלב בשוק העבודה. אף אחד לא יוכל אתכם לעבודה על סמך קריית הספר הזה ופרויקט גמר, מוצר כלשהו. יש כמה נתיבים נוספים שיאפשרו לכם להמשיך למדוד ולצבור ניסיון – עוד לפני העבודה הראשונה.

המחברות להילת הפיתוח

אחרי שבנויתם את הפרויקט שלכם, או במהלך הפיתוח שלו – חשוב גם להתחבר להילת הפיתוח הישראלית. היא חמה ונעימה ויש לא מעט מתקנים שיכלו לסייע לכם להמשיך הלאה. בפייסבוק יש קבוצה פULAה מאוד של מתקני ריאקט שפועלת בעברית וכן לא הטרף אליה ולהשתתף (בענוה הרואה) בדיונים, לקרוא, ללמוד ולהבין לאן כדאי להتقدم, ואם יש פרויקט פתוח שמחכה למתרדים – נסו להטרף אליו.

פגשים ומיטאפים

באטר [meetup.com](https://www.meetup.com) יש לא מעט מידע על מפגשים בחינם שנקראים מיטאפים. במפגשים אלו נפגשים מתקנים ושותפים הרצאות או בונים יחד דברים. יש מפגשים המיועדים למתקני ריאקט וכן לヒרים לקבוצות השונות שמארכנות מפגשים על מנת לדעת על אלו שמתקנים אחרים בסביבתכם ובשעות המתאימות לכם. במפגשים אלו אפשר גם להמשיך ללמידה על פיצרים שונים בריאקט וגם להיפגש עם מתקנים מקטועים, לשאול לעצם ולקבל הכוונה בנוגע להתקדמות בלימוד השפה ובפרויקטים שונים. מדי פעם גם יש באתר הזמנה להاكتונים ולמפגשים שבהם מתקנים בפועל. כדאי מאוד להצטרף לקבוצה צו ולתיכנת לצד מתקנים אמיתיים בליווי הכוונה אמיתית. כמה שעות עבודה ככלו יקדמו אתכם באופן מטורי, גם אם לא ת贔zo בפרס.

האתר Stackoverflow

האתר [Stackoverflow](https://www.stackoverflow.com) הוא שימושי בשתי דרכים – ראשית הוא מכיל תשובות לשאלות רבות על ריאקט וסביר להניח שתיתקלו בו כאשר תחפשו בגוגל שאלות שונות על ריאקט. אבל הוא חשוב גם כי הוא מאפשר לכם לענות על שאלות של מתקנים אחרים, שאולי פחות מנוסים מכם בריאקט בתחום התיאורטי. השתתפות בדיון וניסיון לענות על שאלות של אחרים בריאקט גם יקדמו מאוד את הידע שלכם. פרופיל עשיר ב-WStackoverflow הוא משאוי שראוי לציין בקורות החיים.

תרומת קוד בגייטהאב

לבסוף, אפשר לתרום קוד בגייטהאב לפרויקטים שימושיים בריאקט. למשל, ספריית קומפוננטות מבוססת על ריאקט. בגייטהאב יש לא מעט פרויקטים שימושיים לתרומת קוד. במהלך הפרויקט שבניהם בודאי ראתם ספריות קומפוננטות חסרה להן בדיקת הקומפוננטה הזו שהייתם רוצים. למה שלא תוסיפו אותה? או תתקנו את הדוקומנטציה? או כתבו בדיקה אוטומטית במקומם שבו היא חסра – תרומת הקוד הזו היא ממש בבחינת ניסיון, ואם תבחרו בנתיב הזה ותצליחו בו, תוכלו אפילו להשתלב בקלות בתעשיית התוכנה.

מצד שני, העבודה בגייטהאב מצריכה ידע בגייט, הבנה באנגלית ויכולת לתקשר עם המתכנתים שעובדים בפרויקט ומגיעים מכל העולם. יש גם מיטאפים שבהם מתכנתים אחרים מסבירים איך עובדים עם גייטהאב ותרומות קוד.

כך או כך – מספר זהה הוא הצעד הראשון בכניסה לעולם המרתך של ריאקט ופיתוח צד ללקוח. אני מאחל לכם הצלחה בלימוד וגם בהשתלבות בתחום.

נספח: **PropTypes**

מחבר: רן בר-זיק עבור חברת HoneyBook

התקשרות בין קומפוננטות שונות נעשות באמצעות `props` שמעבירים קלט או, באמצעות פונקציות, פלט. על מנת להשתמש בקומפוננטה, אנו צריכים לדעת באיזה `props` להשתמש. מה קורה אם מי שימוש (`או, במקרה מסוים יותר, צורך`) את הקומפוננטה שלנו לא משתמש-ב-`props` המתאים? הקומפוננטה לא תעבוד או שתעבד באופן לא צפוי. אם אנו כותבים קומפוננטה, החובה שלנו הוא לסמן לאלו שימושים בה (גם אם המשתמשים בה הם אנחנו, בעוד) באיזה `props` אפשר להשתמש. כדי שמי שימוש בקומפוננטה יקבל חיוו מיידי. איך עושים זאת? באמצעות `PropTypes`. מודול שהוא חלק מריאקט עד גרסה 15.5 והחל מהגרסתה זו הוא יצא למודול משלה שnitן להתקינו בנפרד. `PropTypes` הוא הדרך שלנו להציגו איזה `props` אנו נדרש לקבל ומהו סוג. כך למשל, אני יכול להציג על `props` מסוימים שדרישים ועל כלו שהם רק אופציונליים וכמובן לפרט את הסוגים שלהם. כך, אם מי שימוש בקומפוננטה שונח להעיבר `props` מסוימים, הוא יקבל התראה. אם הוא מעביר מחרוזת טקסט ועוד אנו מצפים למספר, הוא מקבל גם כן התראה מאוד ברורה.

למרות ש-`PropTypes` אינם חלק מריאקט, רבים מהתוכננים שפתחים בריект עובדים איתם ומומלץ מאוד לעשות כן.

התקנת `PropTypes`

בדוק כמה ספירת קומפוננטות חיונית, שלמדנו עלייה בעבר, או `Create-React-App`,anno מתקנים את `PropTypes` באמצעות `Node.js` והכלי שלו להתקנת מודולים: `npm`. בהנחה ש-`Node.js` מותקנת במחשבם ואותם משתמשים באפליקציה `Create-React-App`, הכנסו באמצעות הטרמינל לתיקיה הראשית של האפליקציה שלכם והקלידו את הפקודה הבאה:

```
npm install prop-types
```

אם הכל תקין, לאחר נדקה תוכלו לראות בטרמינל חיוי על כך שההתקנה האличה. מהנוקודה זו תוכלו להשתמש ב-`.PropTypes`.

```
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 || | >= 3.6.0-dev || | >= 3.6.0-beta || | >= 3.7.0-dev || | >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.

+ prop-types@15.7.2
updated 1 package and audited 903897 packages in 7.798s
```

השימוש ב-`PropTypes`

השימוש נעשה בקוד שבו אנו כותבים את הקומפוננטה. בין אם מדובר בקומפוננטה מבוססת קלאס או קומפוננטה מבוססת פונקציה. אנו מיבאים את `PropTypes` באמצעות `import`, בדיקון כמו כל ספירה חיונית אחרת שלמדנו עליה ובאמצעות האובייקט שבו מיבאים אנו מצהירים על ה-`props` השונים.

הבה ונדגים באמצעות הקומפוננטה הפешוטה 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 חדשים, אולי שניים. אם אשכח את מה שאני אמרו להעיבר - מה שיקרה הוא שהקומפוננטה לא תעבור כשרה ולי לא יהיה מושג למה. אני אctrck לנbor בקוד כדי להבין מה השتبש.

הבה ונshall use PropType. ראשית ייבוא ואז 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 **ה-prop**. חלק שלישי הוא האם מדבר ב-**prop** שהוא נדרש לפעולת תקינה של הקומפוננטה.

אם אני אנסה להשתמש בקומפוננטה בלי להציגו על `name`. אני מקבל הודעה שגיאה מאוד ברווחה שמודיעה לי על כך ש-`prop name` מסומן כנדרש אך הערך שלו הוא לא מוגדר.

```
✖ Warning: Failed prop type: The prop `name` is marked as required in `Welcome`, but its value is `undefined`.
  in Welcome (at App.js:10)
  in App (at src/index.js:16)
  in Provider (at src/index.js:16)
```

אם אני עביר סוג נתון שאינו תואם לסוג שאני מבקש, אני מקבל גם הודעה שגיאה ברווחה שמודיעה לי ש-`prop name` קיבל ערך שאינו תואם את מה שהקומפוננטה רוצה לקבל.

```
✖ Warning: Failed prop type: Invalid prop `name` of type `number` supplied to `Welcome`, expected `string`.
  in Welcome (at App.js:10)
  in App (at src/index.js:16)
  in Provider (at src/index.js:16)
```

למרות הודעה השגיאה – הקומפוננטה תמשיך לרווח תרגיל ותנסהランדר את הפלט. אם לא יהיו שגיאות נוספות – השגיאות ש-**PropTypes** מציגה לא יגרמו לקומפוננטה לא לרווח.

ערכים שאפשר לקבוע

אנו יכולים להודיע באמצעות `PropTypes` על מגוון רב של סוגי נתונים שונים – כמעט כל הנתונים הprimיטיביים האפשריים. בטבלה זו יש ריכוז של סוגי הנפוצים:

סוג הערך	תיאור	דוגמה
מחרוזת טקסט פשוט	מחרוזת טקסט פשוטה	<code>PropTypes.string</code>
מספר	כל מספר שהוא	<code>PropTypes.number</code>
בוליאני	<code>false</code> או <code>true</code>	<code>PropTypes.bool</code>
פונקציה	כל פונקציה, רגילה או אונומית	<code>PropTypes.func</code>
מערך	מערכות המכילים מידע, כולל מערך ריק	<code>PropTypes.array</code>
אובייקט	אובייקטים שמכילים מידע או מתודות, כולל אובייקטים ריקים	<code>PropTypes.object</code>
אלמנט ריאקט	קומפוננטה ריאקטית כלשהי שימושית	<code>PropTypes.element</code>

אני יכול להחליט שה-`Type` שלי יקבל כמה סוגיים. כך למשל, אם הקומפוננטה שלי יודעת להתחזק עם מספר או טקסט ב-`prop` מסוים. אני יכול לציין שאני יכול או סוג אחד, או סוג שני. אני עושה את זה באמצעות הפקודה `oneOfType` במקומ הסוג.

למשל:

```
import PropTypes from 'prop-types';
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 יוכל לשרש את `isRequired` במידה והפרמטר נדרש. אך זו לא חובה.

חשוב להזכיר `PropTypes`. מדובר בתוספת קטנה וחשובה מאוד לכל קומפוננטה שיכולה לעשות סדר בכך. נכון, זה מעט יותר עבודה, אבל לטווח ארוך זה חשוב וՃאי.

תודה רבה על הרכישה של הספר!

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

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

העתק זהה נמכר ל:

eli.k23@gmail.com

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

תודה וקריאה נעימה!