תודה רבה על הרכישה של הספר!

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

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

העותק הזה נמכר ל:

omer@webiks.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נחבאים פרטי הרוכש באופן שקוף למשתמש. כדאי מאוד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למישהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיחקו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!

ללמוד ריאקט בעברית

רן בר-זיק

Ono
הקריה האקדמית אונו
Ono Academic College
החוג למדעי המחשב

Really Good
elementor
HoneyBook

ללמוד ריאקט בעברית

רן בר-זיק

מהדורה: 2.0.0

כל הזכויות שמורות © רן בר-זיק, 2021.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רישיון לא-בלעדי, לא-ייחודי, אישי, בלתי ניתן להעברה (למעט על פי דין), ובלתי 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 לצטט קטעים קצרים מהספר במסגרת הגנת שימוש הוגן, כלומר פסקה או שתיים, כאשר אתה מפנה למקור ומזכיר את רן בר-זיק כמחבר הספר.

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

עריכה לשונית: יעל ניר

הגהה: חנן קפלן

עיצוב הספר והכריכה: טל סולומון ורדי (tsv.co.il)

הפקה: כריכה – סוכנות לסופרים

www.kricha.co.il

תוכן העניינים

על "ללמוד ריאקט בעברית" 10

על המחבר 11

על העורכים הטכניים 12

דורון זבלבסקי 12

גיל פינק 12

דורון קילזי 13

על החברות התומכות 14

Really Good 14

אלמנטור 14

HoneyBook 16

על ריאקט 17

דרך הלימוד 19

על המונחים בעברית 19

סביבת העבודה הבסיסית 20

סביבת עבודה בסיסית ב-codepen 23

אפליקציית ריאקט 26

בניית קומפוננטת פונקציה בסביבת העבודה הבסיסית 32

בנייה של סביבת ריאקט אמיתית ומורכבת יותר באמצעות Create React App 40

התקנת Node.js על המחשב שלכם 41

התקנה על חלונות 41

התקנה על מק 44

התקנה על לינוקס 45

עבודה עם טרמינל 45

הפעלת הטרמינל 45

ניווט בטרמינל 47

מציאת מיקומים בטרמינל דרך חלונות 48

טרמינל ב-Visual Studio Code 49

בדיקת גרסת Node.js דרך הטרמינל 49

עבודה עם Create React App 50

קשיים ותקלות 55

כתיבת קומפוננטה ראשונה ב-React App Create *57*

Export\ Import *67*

ייבוא מנתיבים אחרים **71**

אין חובה להשתמש בסיומת הקובץ ts **72**

ייצוא של כמה משתנים **72**

ייבוא של תמונות, CSS ומשאבים אחרים **73**

JSX *76*

רשימות ב-JSX **83**

קומפוננטה עם תכונות (props) *91*

דיבאג *105*

סטייט *110*

ריאקט ורנדור **112**

סטייט **113**

תכנון מבנה הקומפוננטות *123*

אירועים ועדכון קומפוננטות *129*

אירועי DOM **130**

אירועים סינתטיים/ ריאקטיים **131**

אלמנט fragment *152*

useEffects *156*

קומפוננטה ללא שם *162*

עיצוב קומפוננטות *165*

CSS בסיסי **167**

סלקטור 168

תכונות 170

קלאס ריאקטי *174*

מעגל החיים בקלאס **184**

componentDidMount 184

componentDidUpdate (prevProps, prevState, snapshot) 184

componentWillUnmount 184

shouldComponentUpdate (nextProps, nextState) 184

getDerivedStateFromError (error) 185

componentDidCatch (error, info) 185

שימוש בקומפוננטות ממקורות אחרים *195*
ייבוא כמה קומפוננטות 200
מודולים חסרים 202
HOC: Higher Order Component *215*
ראוטינג *228*
קונטקסט *251*
חיבור לשרת עם סרוויסים *266*
מבוא לבדיקות עם jest *274*
חלק ראשון – import 276
חלק שני – כתיבת מסגרת הבדיקה 276
חלק שלישי – הרצת הקומפוננטה 276
חלק רביעי – הבדיקה 276
בדיקות לקומפוננטה עם props 279
פונקציית בדיקה – assert 280
not 280
toBeTruthy 280
toBeFalsy 281
toContain 281
יצירת בילד והעלאת האפליקציה לסביבה חיה *286*
סיום – ומה עכשיו? *292*
התחברות לקהילת הפיתוח 293
מפגשים ומיטאפים 293
האתר Stackoverflow 293
תרומת קוד בגיטהאב 294
נספח: PropType *295*
התקנת PropTypes 295
השימוש ב-PropTypes 296
ערכים שאפשר לקבוע 300
נספח: שינויים מהמהדורה הקודמת (מהדורה 1.1.1) *302*
נספח: שינויים ממהדורה 1.0.0 ..... *302*

על "ללמוד ריאקט בעברית"

ריאקט היא ספרייה פופולרית מאוד לפיתוח אתרי אינטרנט, אפליקציות ווב, ובכלל כל סוג של ממשק אינטרנטי. באמצעות ריאקט והאקוסיסטם הנרחב והעשיר שלה, כולל קומפוננטות רבות המלוות אותה, כל מתכנת יכול ליצור אתר או אפליקציה מורכבים במהירות רבה ובאיכות יוצאת דופן. לריאקט יש גם מעטפת (כמו אלקטרון או קורדובה) המאפשרת לממשק שפותח לווב להיות מותאם בקלות גם לתוכנות מחשב של ממש. העושר של האקוסיסטם הפיתוחי של ריאקט וחוזק הקהילה שלה בארץ ובעולם מאפשרים לקבל גם המון-המון מידע וסיוע בכל תקלה ובעיה. אם מתכנת נתקל בבעיה בפיתוח, סביר להניח שהוא ימצא סיוע ומידע בפורומים ובקבוצות מתכנתים. יש גם שפע של מיטאפים וקבוצות דיון המוקדשים למתכנתים העובדים בריאקט ונוסף על כך, הביקוש למתכנתים המכירים את ריאקט הוא גבוה. כל הדברים האלו הופכים את ריאקט לאידיאלית לכניסה לעולם פיתוח צד הלקוח.

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

על המחבר

רן בר-זיק הוא מפתח תוכנה במגוון שפות ופלטפורמות מאז 1996 ועובד כמפתח בכיר במרכזי פיתוח של חברות רב-לאומיות, מ-HPE ועד Verizon, שם הוא מפתח בטכניקות מתקדמות הן בצד הלקוח, הן בצד השרת, ושם דגש על בניית תשתית פיתוח נכונה, על שימוש ב-CI\CD וכמובן על אבטחת מידע.

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

משנת 2008 מפעיל רן את האתר "אינטרנט ישראל" (internet-israel.com), שהוא אתר טכני המכיל מדריכים, מאמרים והסברים על תכנות בעברית ומתעדכן לפחות פעם בשבוע.

רן הוא מחבר הספר "ללמוד ג'אווהסקריפט בעברית", "ללמוד Node.js בעברית", "ללמוד MySQL בעברית" והספר "ללמוד פיתוח ווב מעשי בעברית" ומלמד בקריה האקדמית אונו.

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

על העורכים הטכניים

דורון זבלבסקי

הקריירה של דורון התחילה דווקא בצד השרת, בחברות אבטחת מידע שבהן עבד כמהנדס תוכנה, מוביל טכני ומנהל מקצועי.
את המעבר לעולם הפרונטאנד ביצע בשנת 2014 כאשר הצטרף ל-Applitools, שם הקים את קבוצת פיתוח הווב, וכיום הוא מוביל אותה.
במסגרת חיפושים וניסיונות להחיל עקרונות נכונים של הנדסת תוכנה על קוד פרונטאנד הוא התוודע לריאקט ואימץ אותה בחום.
דורון הקים את קהילת ריאקט בישראל, כולל קבוצת פייסבוק ומיטאפ פופולרי, ואף יזם את כנס ReactNext הראשון בישראל והיה מפיק משותף שלו.
נוסף על כך הוא מרצה על ריאקט, על בדיקות אוטומטיות ועל בניית צוותים מנצחים ותורם מזמנו בשמחה למתחילים המבקשים סיוע בתחום במסגרת מפגשים אישיים וקבוצתיים.

גיל פינק

גיל פינק הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert, Microsoft Developer Technologies MVP והמייסד של חברת sparXys.
כיום הוא מייעץ לחברות ולארגונים שונים, שם הוא מסייע בפיתוח פתרונות מבוססי אינטרנט ו-SPAs. הוא עורך הרצאות וסדנאות ליחידים ולחברות המעוניינים להתמחות בתשתיות, בארכיטקטורה ובפיתוח של מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט (Microsoft Official Course MOC), מחבר משותף של הספר " Pro Single Page Application Development" (Apress) ושותף בארגון הכנס הבינלאומי AngularUP.
לפרטים נוספים על גיל: http://www.gilfink.net

דורון קילזי

דורון חי ונושם פיתוח לרשת זה כעשור.

את דרכו החל ביחידה טכנולוגית מובחרת בחיל המודיעין, שבה שירת כשש 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 מחברות האינטרנט הגדולות בעולם. בימים אלו הוא עוסק, בין היתר, במעבר של החברה מטכנולוגיות ותיקות כגון Angular.js לריאקט ובפיתוח של מערכות מתקדמות בעולם ה-ad tech.

הדחף של דורון ללמוד ולהשתפר גורם לו להמשיך להתמקצע ולהתאהב בכל יום מחדש בעולם ה-js.

על החברות התומכות

Really Good

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

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

אלמנטור

אלמנטור מפתחת פלטפורמת קוד פתוח לבניית אתרים שמשנה את הדרך בה בונים אתרי אינטרנט בשוק המקצועי. אלמנטור מעניק למעצבים את החופש ליצור עמודי אינטרנט ללא צורך בקוד ולמפתחים את החירות לדחוף את הגבולות, לרענן ולהרחיב את המערכת בצורה קלה ומהירה באמצעות API ידידותי למפתחים, ובכך לחסוך זמן פיתוח ולהיות יעילים ורווחיים.

עם מיליוני אתרים הפעילים על אלמנטור וצמיחה חודשית מדהימה, התגבשה סביב הפלטפורמה קהילה חזקה המונה מאות אלפי חברים, מפתחים, משווקים ומעצבים, המקיימים מיטאפים בכל רחבי העולם. מידי יום האלמנטוריסטים מייצרים וצורכים אלפי שעות של הדרכות, סרטי השראה ובלוגים מעמיקים, ומפתחים תורמים קוד ורעיונות באמצעות GitHub. האקוסיסטם המקצועי של אלמנטור מתפתח ללא הפסקה והוא אוצר המוסיף ומעשיר את היכולות של כל יוצר אינטרנט.

באלמנטור אנחנו משתמשים בטכנולוגיות קוד פתוח מתקדמות לפיתוח כלי אינטרנט חדשניים ומהירים. אם גם אתם רוצים להיות חלק מהטכנולוגיה שמשנה את חווית האינטרנט בעולם ויש לכם את הידע כדי לבנות עולם יפה יותר אנחנו מחפשים אתכם, מעצבי UX&UI, מפתחי Full

Stack, מהנדסי Big Data ו DevOps 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 לנהל את כל הלידים בצורה אפקטיבית יותר, ניהול כל התקשרות מול לקוחות הקצה שלהם, ניהול כספים והעברת תשלומים, חתימת חוזים, תזמון פגישות, ניהול משימות, אוטומציה וכו.
הפלטפורמה עוזרת יום-יום לעשרות אלפי אנשים בארה"ב להתנהל בצורה אפקטיבית ומקצועית יותר, כך שהם סוגרים יותר עסקאות בפחות זמן ומאמץ. את הזמן הפנוי שלהם הם יכולים להשקיע בהגדלת העסק, מציאת עוד לקוחות ובמשפחה שלהם.

בהאניבוק הלקוח הוא המרכז ואיתו גם הברנדינג שלו. חשוב לנו לוודא שאנחנו מאפשרים לו להראות הכי טוב שהוא יכול בתקשורת מול הלקוחות שלו. עם טכנולוגיות מתקדמות ודגש על עיצוב, אנחנו מאפשרים לו בקלות לבנות חוזים, הצעות מחיר ואימיילים שנראים טוב ומאפשרים תקשורת מהירה ויעילה עם הלקוחות שלו.

אנחנו מתמודדים עם אתגרים טכנולוגיים, עיצוביים ופיננסיים. כאשר בכל אתגר אנחנו שמים את הלקוח במרכז על מנת להגיע להחלטה נכונה ומהירה. בואו תצטרפו לחברה מצליחה שרוצה לשנות את הדרך בה עצמאים עושים עסקים. חברה שמאפשר ללקוחותיה להתפרנס מהחלום שלהם.
וכבר הזכרנו שהופענו ברשימת מקומות העבודה האטרקטיביים ביותר לשנת 2018 ו 2019? גם בישראל וגם בסן פרנסיסקו (אם תהיתם).

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

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'אווהסקריפט. הדפדפן יודע לקרוא את כל הקבצים האלו ולבנות מהם תמונה שלמה של אתר, ה-HTML קובע את מבנה האתר, ה-CSS והתמונות קובעים את העיצוב שלו וקובצי הג'אווהסקריפט את התנהגותו. בתחילת ימי הרשת כך נראו אתרי אינטרנט; בתחילה ג'אווהסקריפט שימשה לאנימציות או לאינדיקציות שונות בדף ובהמשך לתפקידים מתוחכמים יותר כמו שליחת בקשות באמצעות AJAX. ובכל זאת, באתרי האינטרנט הראשוניים, כל אינטראקציית ניווט כלשהי – לחיצה על כפתור בתפריט או שיגור טופס – שיגרה בקשה לשרת וגרמה לטעינה מחודשת שלו בידי הדפדפן ולקבלת סט חדש של קובצי HTML, CSS וג'אווהסקריפט.

במהלך הזמן החלו להתפתח ספריות ג'אווהסקריפט, כמו jQuery. ספריית jQuery הייתה ספריית עזר שסייעה למתכנתי ג'אווהסקריפט ליצור אפקטים ואנימציות בקלות רבה יותר. קוד הג'אווהסקריפט בצד הלקוח הפך להיות משמעותי יותר ויותר. בשנת 2008 יצאה לשוק ספרייה שנקראת Backbone.js. זו הייתה ספרייה ששינתה לחלוטין את הדרך שבה אנו מסתכלים על צד הלקוח: במקום קוד שמגדיר רק התנהגות – אפליקציה שלמה שיושבת בצד הלקוח ומתנהגת כמו אפליקציית צד שרת, כולל אפשרויות ניווט בעמודים, הבאת מידע מ-API של שרתים ועוד שימושים רבים. היתרונות של שימוש בספריות גדולות כאלו, או יותר נכון פריימוורקים המגדירים את ההתנהגות בצד הלקוח, היו רבים – מקלות פיתוח ועד חוויית שימוש יוצאת דופן עבור הלקוח. הפריימוורקים האלו אפשרו ליצור Single Page Application – SPA. כלומר, כשגולשים באתר ובמעבר בין דפים אין טעינה מחדש מהשרת אלא מעבר חלק בין דף לדף באמצעות רכיב מבוסס ג'אווהסקריפט שנקרא ראוטר (ועליו נלמד בהמשך הספר). Backbone.js הייתה הראשונה, אבל מהר מאוד הגיעו ספריות נוספות כמו Ember.js וכמובן אנגולר. ריאקט הגיעה אחרי אנגולר וגרסתה הראשונה יצאה ביוני 2013. מאז היא תפסה תאוצה משמעותית והפכה לאחת מהספריות הגדולות והנפוצות בעולם.

ריאקט (React) היא ספרייה מבוססת ג'אווהסקריפט המיועדת לצד הלקוח. היא מאפשרת לנו ליצור אתרים שלמים ומערכות שלמות בקלות רבה ובדרך מודרנית ופשוטה. באמצעות ריאקט אנו

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

במקור, המפתחת של ריאקט היא חברת פייסבוק, שעד היום היא התומכת הראשית שלה והמפתחים שלה מובילים את פיתוח הליבה של ריאקט ומתווים את הדרך. ריאקט מפותחת ברישיון קוד פתוח מלא (החל מגרסה 16) וקוד המקור שלה נמצא בגיטהאב. אפשר להשתמש בה לכל שימוש בצורה חופשית. אחד היתרונות הגדולים בריאקט הוא עושר הקומפוננטות שמשתמשות בה, מה שאומר שאפשר ליצור בקלות אפליקציות מורכבות באמצעות שפע הקומפוננטות שמפתחים אחרים פיתחו – דבר המאפשר לכל צוות פיתוח שבוחר בריאקט גמישות ועבודה מהירה מאוד. גם סביבת הפיתוח של ריאקט וכלי הבדיקות שלה, החיוניים לפיתוח בקנה מידה גדול, הם מצוינים ועמידים מאוד. יש לה כמובן גם חסרונות – החיסרון העיקרי הוא שריאקט לא קובעת עבור המפתח את הרכיבים שאיתם הוא יכול לעבוד, דבר שעלול להוביל לבלבול או לקבלת החלטות לא נכונות, אבל יש מפתחים שיראו בזה יתרון. כך או כך, נכון לזמן כתיבת הספר, רוב המתכנתים שצריכים לפתח אתר כלשהו בצד הלקוח בוחרים בריאקט.

בשנים האחרונות, טייפסקריפט היא הבחירה הראשונה בכל הנוגע לשימוש בריאקט. היכולת של טייפסקריפט למנוע תקלות שונות הקשורות לסוגי מידע וקלות השימוש בה גרמה לטייפסקריפט להיות נפוצה מאד. זו הסיבה שבספר אנו לומדים ריאקט עם טייפסקריפט, למרות שניתן לכתוב ריאקט עם ג'אווהסקריפט בלבד.

הקונספציה של ריאקט ודרך העבודה בה לא שונות מהותית מספריות אחרות כמו אנגולר או Vue. אז אם אינכם מכירים אף פריימוורק או ספרייה של ג'אווהסקריפט, ריאקט היא מקום מצוין להתחיל בו את המסע לעולם המופלא של פיתוח צד לקוח. זאת אף שריאקט שונה בכמה דרכים מהותיות, שאותן נלמד בהמשך, מספריות אחרות כמו אנגולר.

דרך הלימוד

הניסיון שלי מלמד שכל דבר חדש בתכנות לומדים דרך הידיים. אני ממליץ מאוד להעתיק כל דוגמה וכל קטע קוד בספר, להדביק אותם ב-IDE החביב עליכם – 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 ובהסברים. כדאי מאוד לא לוותר ולא להרפות ולנסות שוב ושוב עד שמבינים את הפתרון.

יכול להיות שלמרות ההסברים ולמרות הדוגמאות לא תבינו נושא מסוים או שלא תבינו אותו עד הסוף, לעומק. זה קורה לטובים ולמבריקים ביותר. הפתרון? חיפוש בגוגל – במיוחד באנגלית. כיוון שריאקט היא כל כך פופולרית, יש סיכוי סביר ביותר שמישהו כבר נתקל בבעיה הזו וכתב עליה משהו. אתרים כמו StackOverflow והפורומים השונים מכילים שפע של מידע ותשובות לשאלות שונות. נוסף על כך, בפייסבוק יש לא מעט קבוצות מקצועיות בעברית שישמחו לסייע לכם – בפרק הסיכום של הספר יש כמה קישורים רלוונטיים.

כמובן שדרך טובה מאד ללמוד היא באמצעות בינה מלאכותית, כמו למשל צ'אט GPT. הבינה המלאכותית הזו זמינה באתר https://chat.openai.com וניתן, באמצעות תיבת השיח שם, לשאול שם שאלות או להדביק קוד שלם או שגיאות שונות ולשאול את הבינה המלאכותית מה מקורן ואיך לפתור אותן.
למרות הפיתוי העז, אני ממליץ לא להעזר יותר מדי בבינה מלאכותית לכתיבת קוד מורכב אלא להתאמץ ולכתוב את הקוד בעצמכם. בעבודה היומיומית עם בינה מלאכותית יש חשיבות רבה לידע בקוד המאפשר גם לבדוק את הקוד המוצע על ידי הבינה המלאכותית וגם להנחות את הבינה המלאכותית בביצוע משימות שונות. ניתן לרכוש את הידע הזה רק בעבודה עצמית.

על המונחים בעברית

אני כותב בעברית על טכנולוגיה ועל תכנות כבר יותר מעשור, והדילמה באילו מונחים בעברית להשתמש מלווה אותי תמיד. מצד אחד, האקדמיה ללשון העברית מספקת לנו מונחים רבים בעברית. מצד שני, בתעשיית ההייטק, שממנה אני מגיע, איש לא משתמש ברבים מהמונחים האלו. אם תגיעו לריאיון עבודה ותגידו: "במפגש המתכנתים האחרון שמעתי על דרך חדשה לבצע

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

חשוב לציין שאיני בז כלל לאקדמיה ללשון ושחלק מהמונחים שלה אכן נכנסו לשפה המדוברת במרכזי הטכנולוגיה השונים (למשל: קובץ או מסד נתונים), אבל בכל מקום שהייתה בידיי הבחירה בין להיות מובן לבין לעמוד בכללי הלשון, העדפתי להיות מובן.

סביבת העבודה הבסיסית

זה הפרק החשוב ביותר, כיוון שאי-אפשר ללמוד קוד בלי ללכלך את הידיים בקוד משלכם. קריטי לקרוא את הפרק הזה וליצור סביבת עבודה בסיסית על המחשב שלכם. אם אתם נתקלים בקושי כאן – אל תוותרו ואל תתייאשו. התקנת סביבת עבודה היא החלק הקשה ביותר בלימוד טכנולוגיה חדשה. נסו שוב ושוב – בצעו ריסטרט למחשב, נסו ממחשב אחר, שדרגו את מערכת ההפעלה, נסו מדפדפן אחר שאינו הדפדפן הרגיל שלכם – כל טכניקה וטכניקה. פשוט אי-אפשר לדלג על השלב הזה. נסו להעזר בכלי בינה מלאכותית ולהזין לתוכם את השגיאות, אם יש כאלו, כדי לקבל סיוע. זה החלק החשוב ביותר.

ריאקט מורכבת מכמה חלקים – כולם בקוד ג'אווהסקריפט. החלק הראשון הוא הספרייה עצמה: ריאקט. מדובר בקובץ ג'אווהסקריפט מרכזי המכיל את כל הפונקציונליות של הספרייה. הקובץ מכיל מודולים של ג'אווהסקריפט וגם משתנים גלובליים ואחרים – ובלעדיו אי-אפשר להשתמש בריאקט. כשאנו בונים סביבת עבודה בסיסית, אנו נזקקים לריאקט. הגיוני, לא?
החלק השני הוא react-dom. גם הוא חלק מהספרייה וגם הוא כתוב בג'אווהסקריפט. הוא מכיל את הפונקציות הקושרות בין ריאקט ל-DOM. ראשי התיבות של DOM הם Document Object Model, ואני ארחיב לגביו בהמשך. כרגע צריך פשוט לזכור שמדובר בעוד חלק של הפריימוורק שאנו צריכים איתנו כשאנחנו מפתחים עבור אתרי אינטרנט.

החלק השלישי הוא babel. מה זה? מדובר בספריית ג'אווהסקריפט שימושית ופופולרית מאוד. יש לה כמה תפקידים חשובים. במקור babel סייעה למתכנתים שרצו להתאים את קוד הג'אווהסקריפט שלהם לדפדפנים ישנים שלא תמכו בפיצ'רים החדשים של השפה, למשל דפדפנים שלא ידעו מה זה let או const. הספרייה הזו לקחה את כל הקוד המודרני של ג'אווהסקריפט והעבירה אותו תהליך, שבמסגרתו הוא הפך לקוד שתואם גם דפדפנים ישנים. למשל, היא המירה את let ל-var, כך שדפדפן ישן יוכל לעבוד איתו. התהליך הזה נקרא "טרנספירציה" – זו המילה במדעי המחשב שמשמעה לקחת קוד שכתוב בשפה מסוימת ולתרגם אותו לקוד שכתוב בשפה אחרת. במקרה הזה לקחת קוד שכתוב בג'אווהסקריפט מודרנית ולהעביר אותו לקוד שכתוב בג'אווהסקריפט מגרסאות קדומות.

במקרה שלנו, ל-babel יש תפקיד משמעותי בהמרת ה-JSX שלנו, שהוא הסינטקס שבו אנו כותבים בריאקט קומפוננטות לקוד HTML שהדפדפן יודע לעבוד איתו. על JSX נלמד בהרחבה בהמשך.
אלו החלקים הבסיסיים שחייבים להיות בסביבת העבודה שלנו. נוסף על הקוד שלנו, אנו צריכים ליצור דף HTML ריק לחלוטין שקורא באמצעות ה-src לשלושת הקבצים האלו וגם לקובץ שבו אנו כותבים את הקוד שלנו.

אנו יכולים להוריד את הקבצים האלו מהאתר של ריאקט או להשתמש בהם ישירות מ-CDN. ראשי התיבות של CDN הם Content Delivery Network – זהו כינוי לשרתים גדולים שנמצאים בכל מקום בעולם. חלק מהם מציעים שירות פתוח לציבור של אחסון קבצים של ספריות גדולות ומובן שריאקט, הפריימוורק הפופולרי בעולם, נמצאת ביניהן. ה-CDN שהדוקומנטציה של ריאקט משתמשת בו הוא unpkg ואנו נשתמש בו. unpkg מציעה את הקבצים בקישורים הבאים:

קובץ הפריימוורק של ריאקט:

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

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

קובץ react-dom:

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

קובץ ה-babel:

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

כך קובץ ה-HTML שלנו נראה:

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

בתחתית הקובץ נמצא הקוד שלנו. אנו מסמנים ל-babel שהוא קוד שצריך לעבור טרנספילציה באמצעות הצמדה של:

```
type="text/babel"
```

לתגית ה-script. בין התגית הפותחת של ה-script לתגית הסוגרת שלו, אנו יכולים לכתוב את קוד הג'אווהסקריפט שלנו. כרגע יש שם את השורה הבאה:

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

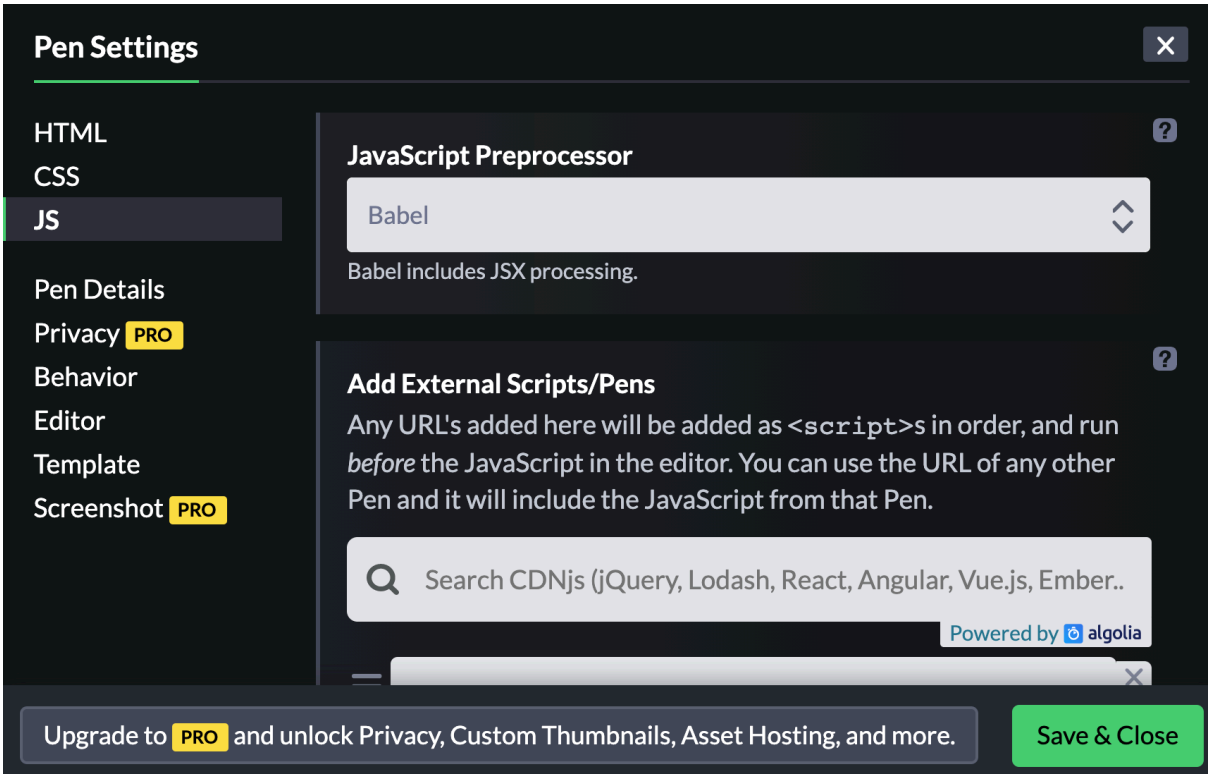
באמצעות עורך הקוד החביב עליכם (כמו Visual Studio Code החינמי), צרו את קובץ ה-HTML הזה ושמרו אותו במחשב המקומי שלכם. פתחו את קובץ ה-HTML בדפדפן באמצעות כניסה לתיקייה שבה הקובץ נשמר ולחיצה על "פתח עם דפדפן" וּוַדאו שהוא עולה ושבקונסולה של כלי המפתחים מופיע Here will be React code. אם כן – זהו, אנחנו מוכנים לעבודה ראשונית.

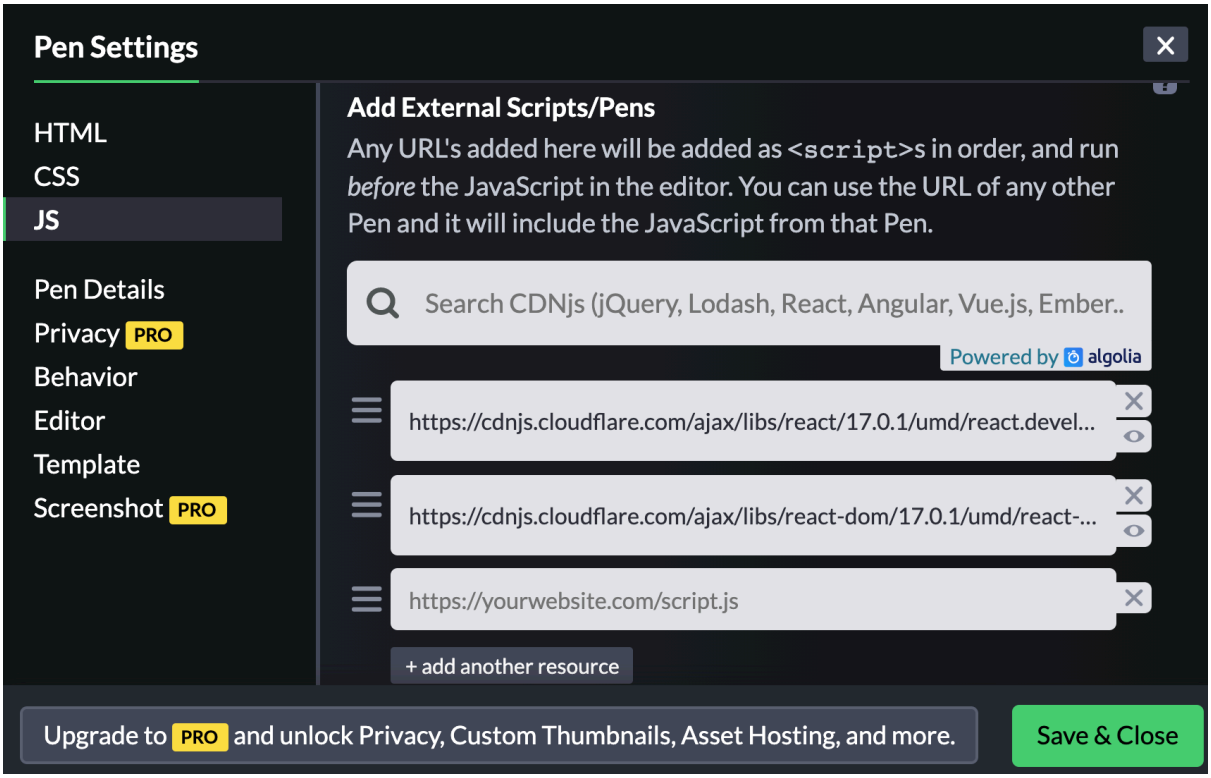
כדאי לשים לב שאנו עדיין לא משלבים טייפסקריפט בקוד, אנו נשלב אותו בסביבת העבודה המתקדמת. כרגע אנו רק לומדים איך ריאקט עובד ולא רוצים לערב טייפסקריפט.

סביבת עבודה בסיסית ב-codepen

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

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





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

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

צפו בקונסולה וראו את המסר Here will be React code.

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

פרק 1

אפליקציית ריאקט

אפליקציית ריאקט

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

כדי ליצור אפליקציית ריאקט מתחת לאלמנט מסוים אנו חייבים פשוט... לבחור אותו. אנו נגדיר div, שזה אלמנט HTML פשוט ביותר עם id פשוט. להזכירכם – id הוא סלקטור ייחודי המגדיר אלמנטים ב-HTML שאנו בוחרים לתת להם זהות ספציפית.

בקובץ ה-HTML שלנו כבר יש הגדרה של div כזה:

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

הינה ההגדרה – div שה-id שלו הוא content:

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

זה המקום שבו תוצב אפליקציית הריאקט שלנו. אנו צריכים רק לזכור שה-id הוא content.

את אפליקציית הריאקט אנו יוצרים באמצעות:

```
ReactDOM.createRoot
```

המתודה הזו באה עם האובייקט הגלובלי ReactDOM. האובייקט הגלובלי הזה בא בזכות קובץ ה-react-dom.js, שדאגנו שיהיה בסביבת הפיתוח שלנו. המתודה מקבלת ארגומנט אחד שהוא האלמנט שאנו בוחרים כדי להריץ בו את ה-JSX של האפליקציה ומחזירה אלמנט ריאקטי שמייצג את האפליקציה. בו יש את מתודת render שמקבלת את ה-JSX.

הציבו את הקוד הזה בין תגיות הסקריפט שלכם:

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

שמרו וצפו בתוצאה. אתם תראו Hello World! בדפדפן אם הכול תקין.

אם לא הכול תקין ואתם לא רואים Hello World! הפסיקו לקרוא, פתחו את הקונסולה, צפו בשגיאות וחפשו אותן בגוגל כדי לתקנן, ודאו שהעתקתם את הקוד כשורה והמשיכו רק כאשר אתם יודעים שסביבת הקוד שלכם עובדת.

הבה נעבור על הארגומנטים השונים של ReactDom. נתחיל מהשני, ה-target. אני יוצר רפרנס לאלמנט שבו אני רוצה להציב את התוכן שלי באמצעות:

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

אני מעביר אותו לקבוע rootElement. אתם אמורים להכיר את זה אם יש לכם ידע בג'אווהסקריפט. אם לא, אנא חזרו על החומר של ג'אווהסקריפט ו-HTML (שנמצא גם בספר שלי "ללמוד ג'אווהסקריפט בעברית"). את הקבוע rootElement אני מעביר כארגומנט אל הפונקציה createRoot של ReactDOM שמחזירה לי אלמנט שנכנס אל הקבוע root. מעכשיו בעצם root הוא אפליקצית הריאקט שלי ואני יכול לקרוא למתודת render ולהכניס לתוכה ארגומנט של JSX.

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

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

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

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

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

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

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

תרגיל

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

פתרון

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

ראשית ניצור קובץ HTML פשוט שבו יש קריאה לשלושת הקבצים שאנו צריכים לאפליקציית ריאקט ושעליהם הסברנו בתחילת הפרק.

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

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

השלב הבא הוא ליצור את אפליקציית הריאקט באמצעות:

```
ReactDOM.createRoot
```

המתודה הזו מקבלת ארגומנט שהוא האלמנט שהאפליקציה תהיה בו. אני מקבל אותו באמצעות getElementById, שהוא מתודת ג'אווהסקריפט שנמצאת בדפדפן באופן טבעי (בלי קשר לריאקט), ומעביר אותו באמצעות משתנה:

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

המתודה מחזירה את ה-root של האפליקציה. אובייקט מיוחד שיש בו מתודות של אפליקצית ריאקט.

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

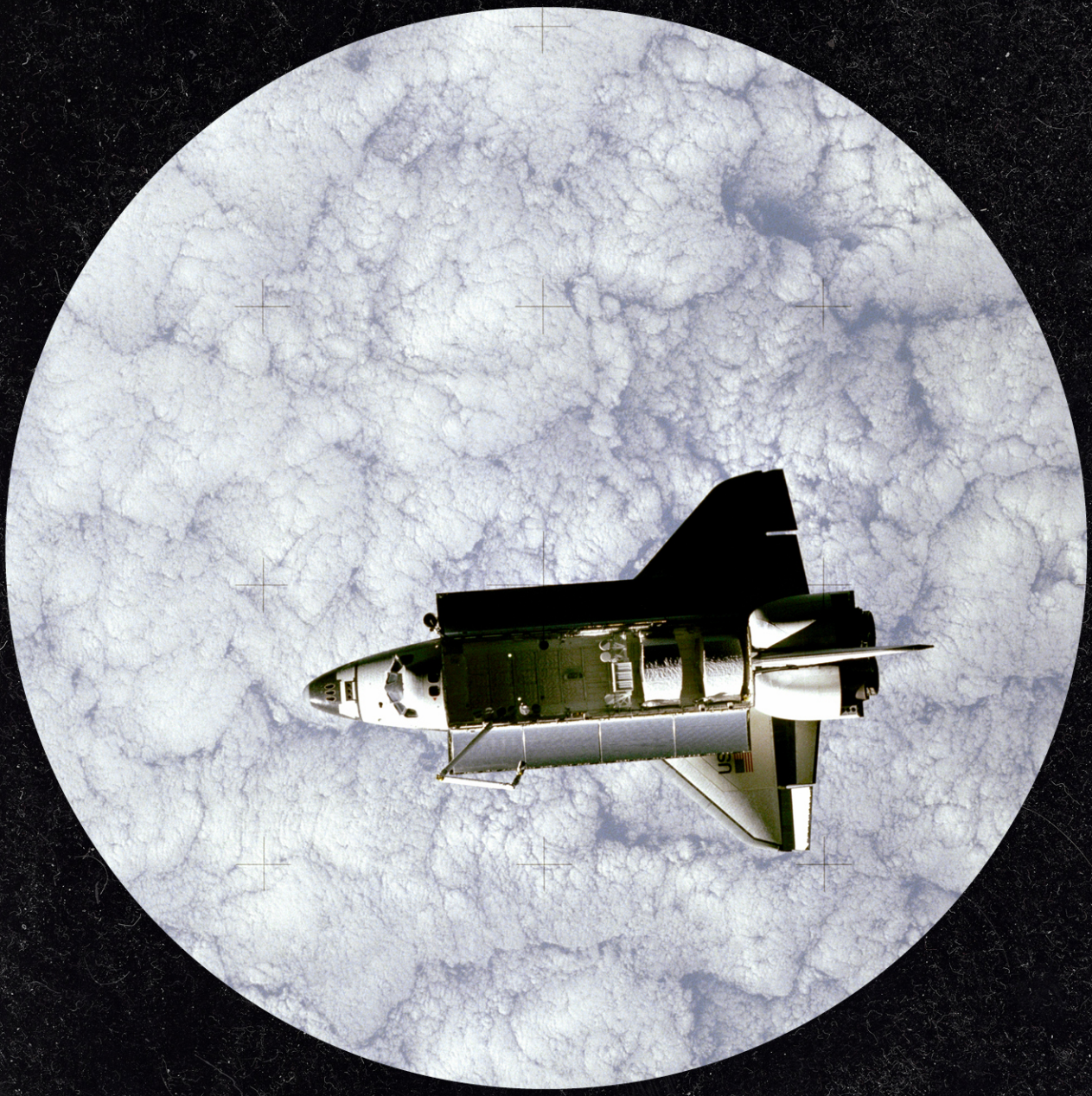
```
root.render(
   <h1>Hello World!</h1>
);
```

שמירת ה-HTML ופתיחה שלו באמצעות הדפדפן תציג לי Hello World!. או כל טקסט אחר.

otherworld.co.il

פרק 2

בניית קומפוננטת פונקציה בסביבת העבודה הבסיסית



בניית קומפוננטת פונקציה בסביבת העבודה הבסיסית

אחרי שלמדנו לבנות את האפליקציה ואפילו למדנו על JSX בסיסי, הגיע הזמן ללמוד על המבנה הבסיסי של ריאקט – קומפוננטות. אחת המהפכות הגדולות בפיתוח צד לקוח שריאקט הביאה היא הקומפוננטות. מדובר ביחידות תוכנה קטנות שאפשר להשתמש בהן שוב ושוב במקומות שונים באתר שלנו. המתכנתים בריאקט יוצרים למשל קומפוננטה אחת של טבלה הניתנת למיון ויכולים להשתמש בה בכל מקום באפליקציה או באתר שלהם. אם הם צריכים לשנות את הטבלה, הם משנים קומפוננטה אחת בלבד. בכל אפליקציית ריאקט יכולות להיות אינספור קומפוננטות. למתכנתים חדשים קל לחשוב על קומפוננטות בתור פונקציות.

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

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

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

הבה ניצור קומפוננטה שהיא זו שתדפיס עבורנו את:

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

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

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

כדאי לשים לב לכמה דברים – ראשית, שם הפונקציה מתחיל באות גדולה. זו קונבנציה של ריאקט שמחייבת כל קומפוננטה שהיא. שנית, הפונקציה מחזירה JSX. במקרה הזה מדובר ב-HTML פשוט. זה מאפשר לנו להבדיל בקלות בין פונקציה רגילה לבין פונקציה שיוצרת קומפוננטה.
איך נשתמש בקומפוננטה הזו? בדיוק כמו HTML. כך:

```
<Greeting />
```

כאלמנט שסוגר את עצמו. או כך:

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

אף על פי שהקונבנציה החד-משמעית היא לכתוב אלמנט שסוגר את עצמו.

ואיך הקוד המלא שלנו ייראה? כך:

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

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

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

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

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

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

בתוך הקומפוננטה אני יכול להשתמש בעוד קומפוננטות, וזה חשוב מאוד. הבה ניצור קומפוננטה נוספת שבה נשתמש בתוך קומפוננטת Greeting. משהו בסגנון הזה:

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

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

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

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

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

ניתן להשתמש בקומפוננטה הזו בכל מקום – היישר באפליקציה או בתוך כל קומפננטה אחרת. במקרה הזה מי שמשתמש בה הוא קומפוננטת Greeting. איך היא משתמשת בה? בדיוק כמו ב-HTML. שם הקומפוננטה כתגית HTML:

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

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

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

תרגיל:

צרו אפליקציית 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!

פתרון:

```
function Inigo() {
  return <span>Hello, my name is Inigo Montoya</span>
}
function Greeting() {
  return <span>prepare to die!</span>
}

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


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

יצרתי שלוש קומפוננטות פשוטות. הראשונה והשנייה Inigo ו-Greeting מחזירות JSX פשוט. כדאי לשים לב שב-JSX אני חייב לשים אלמנט אב אחד, במקרה הזה בחרתי ב-span.
הקומפוננטות האלו הן קומפוננטות מסוג פונקציה. כדאי לשים לב ששמן חייב להתחיל באות גדולה. מהרגע שהגדרתי אותן, אני יכול להשתמש בהן בכל קומפוננטה אחרת בדיוק כמו כל אלמנט HTML אחר.

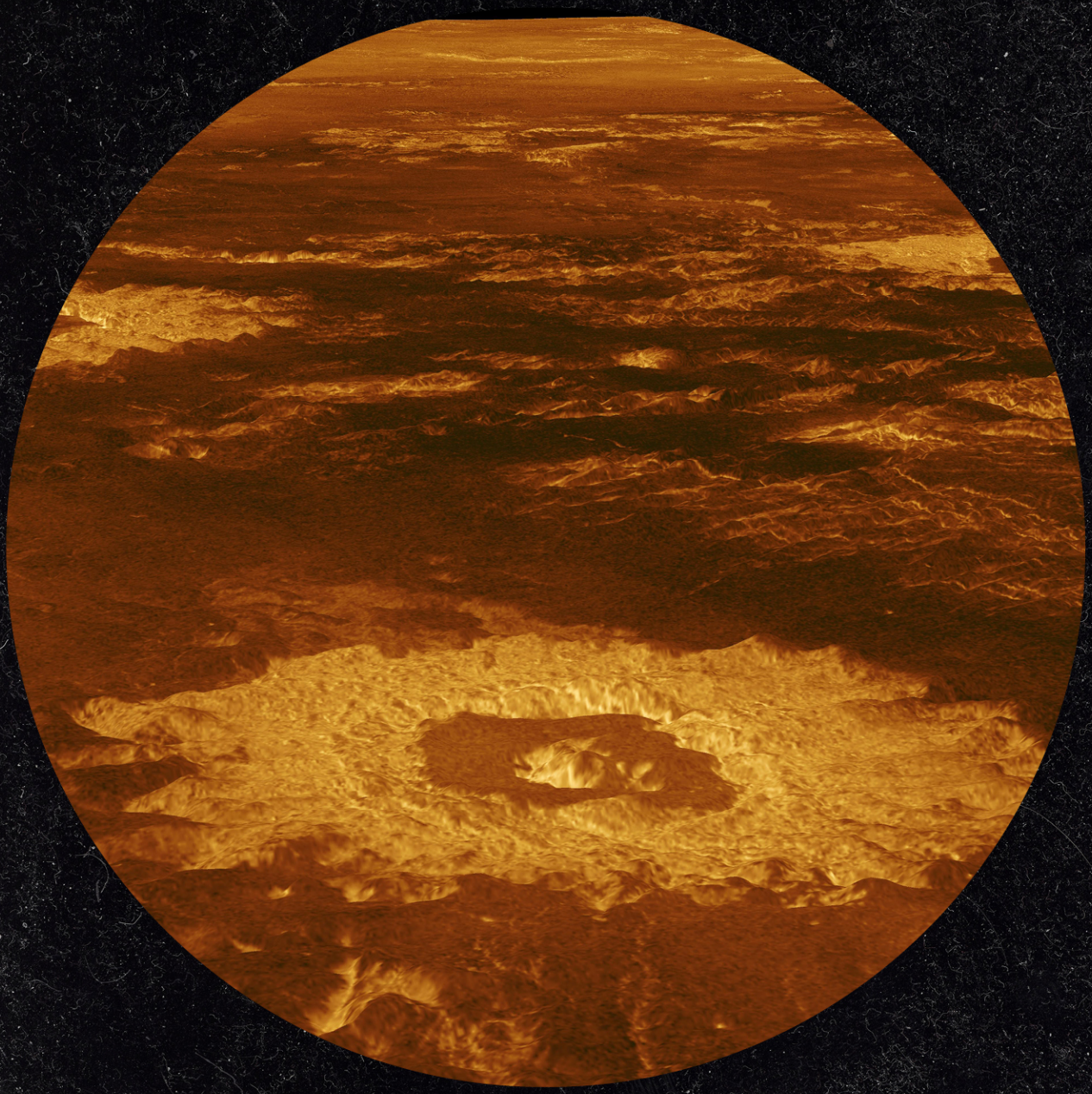
הקומפוננטה השלישית היא קומפוננטת Root והיא מכילה את שתי הקומפוננטות Inigo ו-Greeting. גם פה, חשוב לציין, יש אלמנט אב אחד. גם הוא span.

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

offer@webit.com

פרק 3

בנייה של סביבת ריאקט אמיתית ומורכבת יותר באמצעות CREATE REACT APP



בנייה של סביבת ריאקט אמיתית ומורכבת יותר באמצעות Create React App

בפרקים הקודמים למדנו להקים סביבת עבודה פשוטה וראינו איך יוצרים אתרים או אפליקציות ווב באמצעות אפליקציית ריאקט וקומפוננטות פשוטות. סביבת העבודה 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 שרת מובנה, במקום לטעון את הקובץ דרך מערכת הקבצים המקומית, או hot reload, שמרפרש אוטומטית את האפליקציה בסביבת הפיתוח בכל פעם שאנחנו עושים שינוי, או דיבאגר מובנה... כל אלו ועוד קיימים ממש מן הקופסה בריאקט.

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

בנוסף, ניתן להשתמש בטייפסקריפט בקלות ב-Create React App ולכתוב בה והאפליקציה תדאג להמיר את טייפסקריפט לג'אווהסקריפט יחד עם ה-JSX. זה מאפשר לי להשתמש בטייפים ומאד מקובל.

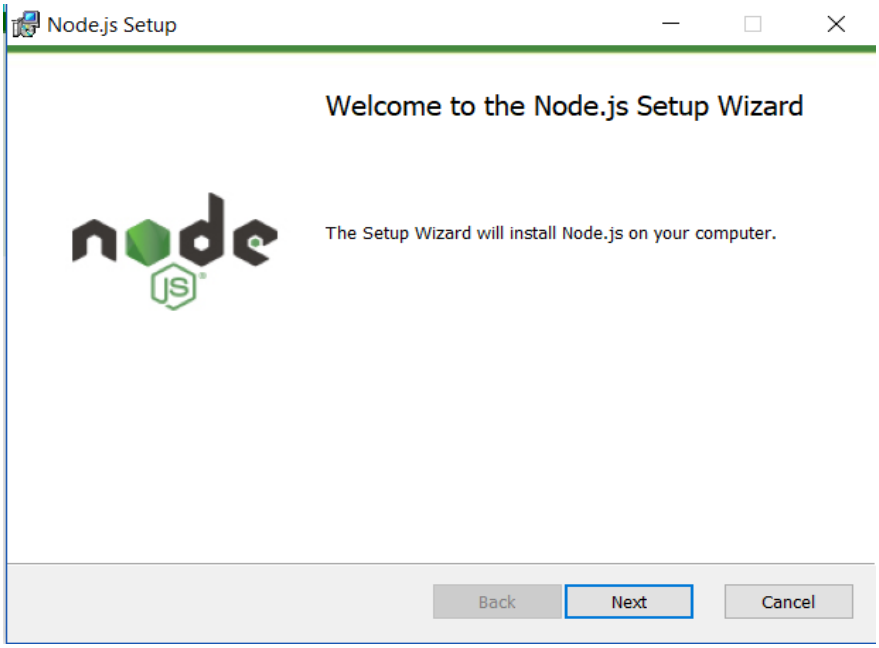
אנו לא יכולים להישאר בעולם ה-Hello World, ולימוד בניית סביבה אמיתית עם Create React App הוא חלק בלתי נפרד מהלימוד של ריאקט. הפרק הזה הוא אחד הפרקים הכי חשובים והכי מורכבים שיש בספר וחשוב מאוד לא להירתע ולא להיבהל. אם יש תקלות, כדאי מאוד לפתור אותן. כאמור, העולם של ריאקט הוא עשיר מאוד ומישהו אחר כבר חווה כל תקלה שתחוו בדרך (אם תחוו). חיפוש מהיר של התקלה בגוגל יסייע לכם מאוד.

התקנת Node.js על המחשב שלכם

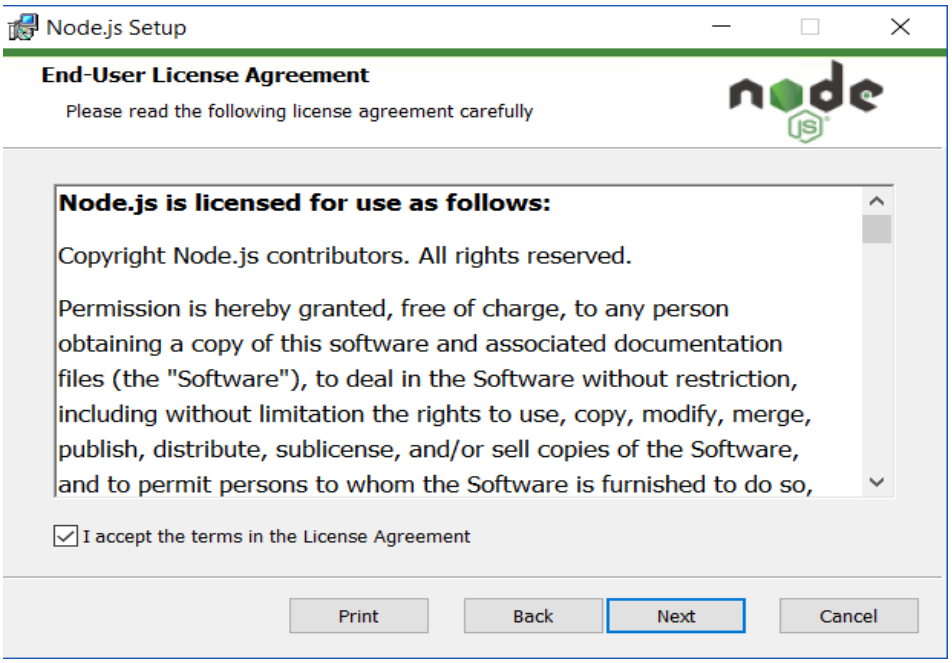
ראשית, אתם חייבים להתקין את Node.js על המחשב המקומי שאתם עובדים בו. Node.js הוא סביבה המאפשרת להריץ ג'אווהסקריפט על גבי מערכת ההפעלה, במקרה הזה מערכת ההפעלה שלכם. ג'אווהסקריפט, למי שלא יודע, היא שפה גמישה מאוד ואפשר להפעיל אותה לא רק בסביבת הדפדפן, כמו שאנו עושים בריאקט, אלא גם בסביבת מערכת הפעלה/ שרתים – ואת זה עושים באמצעות Node.js. הפלטפורמה הזו ניתנת להתקנה בכל מערכת הפעלה שהיא ובקלות.
בחרו את מערכת ההפעלה שלכם והתקינו את Node.js לפי ההוראות.

התקנה על חלונות

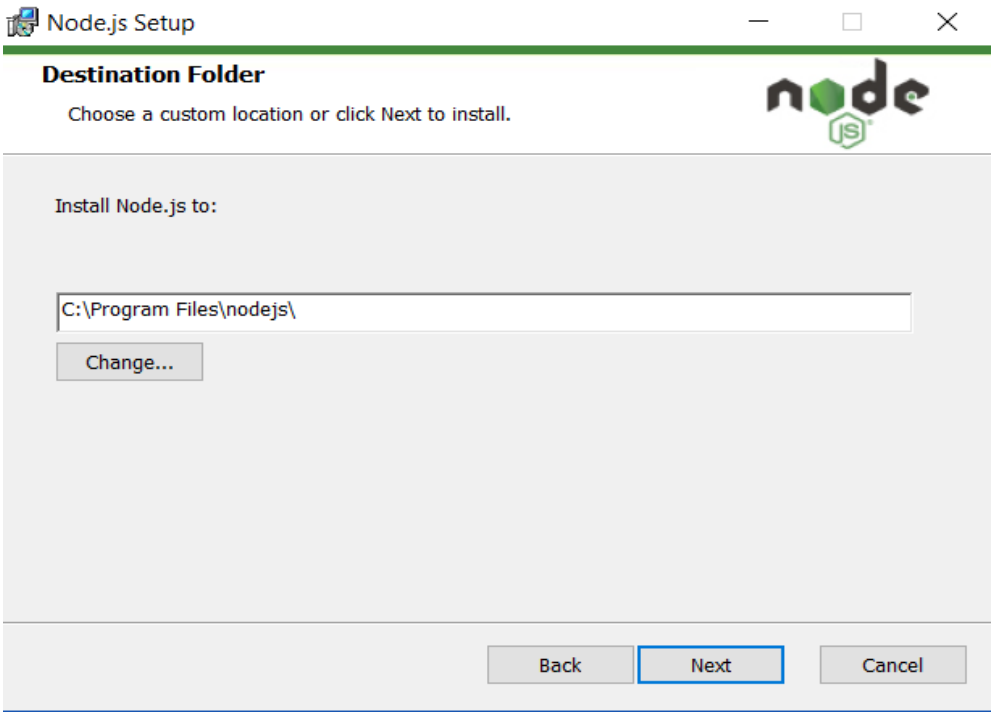
ההתקנה של Node.js על חלונות היא פשוטה. נקליד בגוגל Node.js Download או ניכנס אל:
https://nodejs.org/en/download/
אנו נבחר בגרסת LTS – ראשי תיבות של "גרסה לטווח ארוך", ונבחר במערכת ההפעלה שלנו – אם מדובר בחלונות, יש לנו installer נוח. מורידים, לוחצים על התוכנה שיורדת:



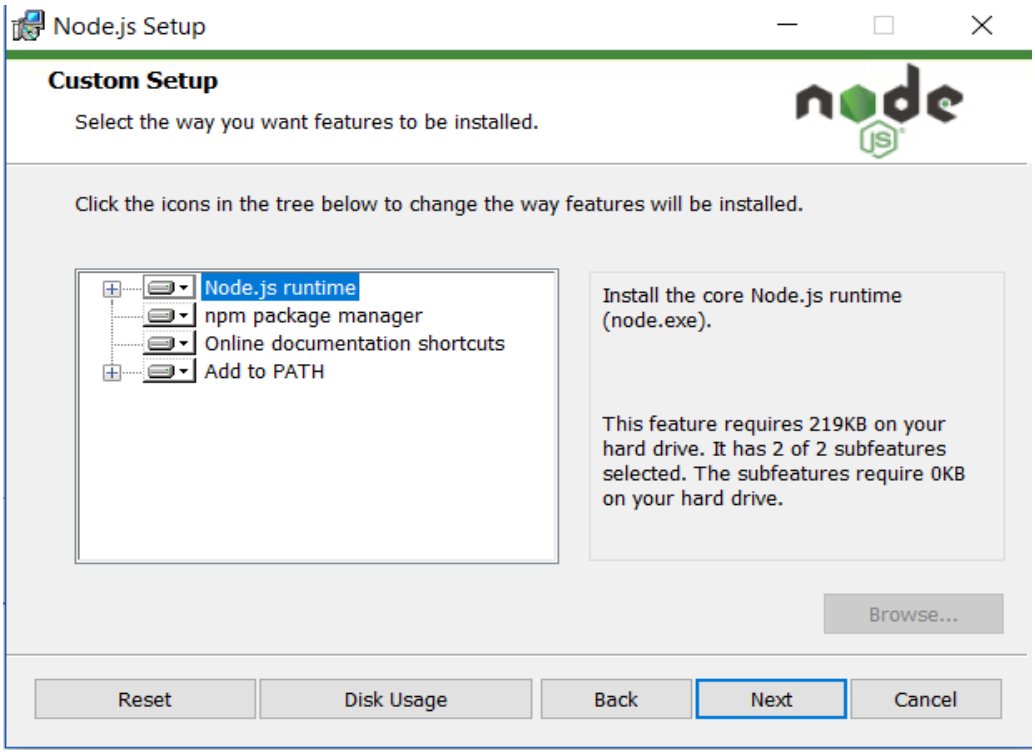
מקבלים את התנאים:



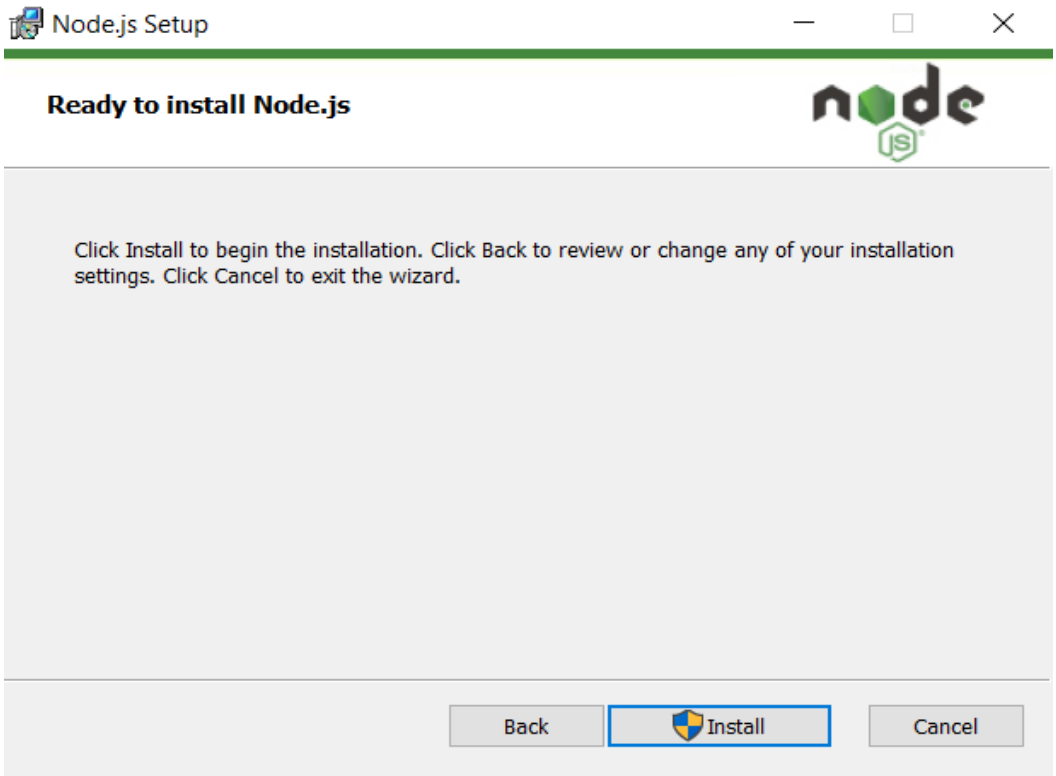
לוחצים על Next:



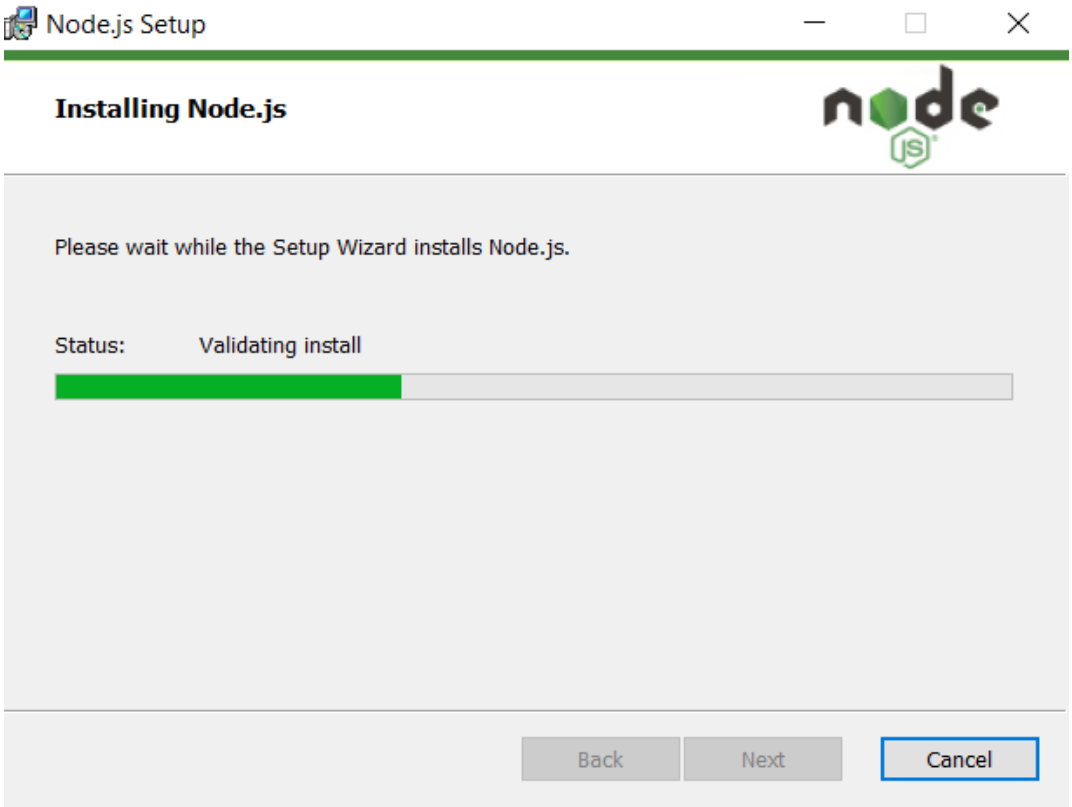
ושוב על Next:



לחיצה על Install לבסוף תתקין את התוכנה:



כל מה שנותר הוא לחכות לסוף ההתקנה:



התקנה על מק

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

אנו נבחר בגרסת LTS – ראשי תיבות של "גרסה לטווח ארוך", ונבחר במק – יֵרד קובץ dmg שאותו אפשר להתקין כמו כל תוכנה אחרת בהנחה שהמחשב שלכם הוא לא מחשב ארגוני שמונע התקנות מהאינטרנט. ההתקנה היא פשוטה ביותר.

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

התקנה על לינוקס

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

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

עבודה עם טרמינל

כמו כל אפליקציית Node.js, גם Create React App מצריכה אותנו לעבוד עם טרמינל. טרמינל הוא הממשק שבו אני מקליד פקודות למערכת ההפעלה. כל מתכנת עובד עם טרמינל כאשר הוא מתחבר מרחוק לשרתים שונים וזה ידע שימושי למדי. אנו נלמד כעת איך להתחבר לטרמינל ואיך לעבוד איתו בחלונות. אני לא מסביר על טרמינל במק או בלינוקס כי אני יוצא מנקודת הנחה שמי שיש לו את מערכות ההפעלה האלו יודע איך להשתמש באופן בסיסי בטרמינל.

הפעלת הטרמינל

בחלונות הגישה לטרמינל פשוטה למדי. בחלונות 10 וב-11 לוחצים על הזכוכית המגדלת, מקלידים cmd ואז לוחצים על אנטר.

מיד מקבלים מסך שחור. לוותיקים בינינו הוא יזכיר את מסך ה-DOS הישן של לפני 20 שנה.

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

C:\Users\barzik>
```

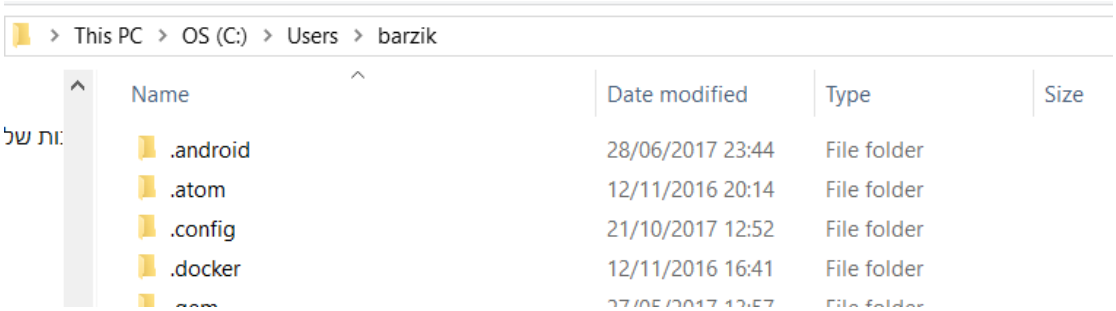
זה הטרמינל של חלונות. לפני כמה עשורים, כך נראו מחשבים. זה המחשב שלכם, אבל הממשק הוא לא גרפי אלא טקסטואלי. יכול להיות שזה ייראה לכם מיושן ולא רלוונטי, בטח ובטח לעומת הממשקים האחרים שאתם מכירים, אבל ככה רוב המתכנתים עובדים – מול הטרמינל של חלונות, של מק או של לינוקס. חשוב מאוד להבין איך עובדים איתו.

כשתיכנסו לטרמינל, מצד שמאל תוכלו לראות את המיקום שלכם. אצלי 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 שצריך, תראו משהו כזה:

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

 Directory of C:\Users\barzik

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

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



ניווט בטרמינל

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

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

אפשר "לעלות" לתיקייה אחת למעלה באמצעות cd ..

נסו לעלות שוב ושוב עד שתגיעו לתיקייה הראשית, הלוא היא c:.

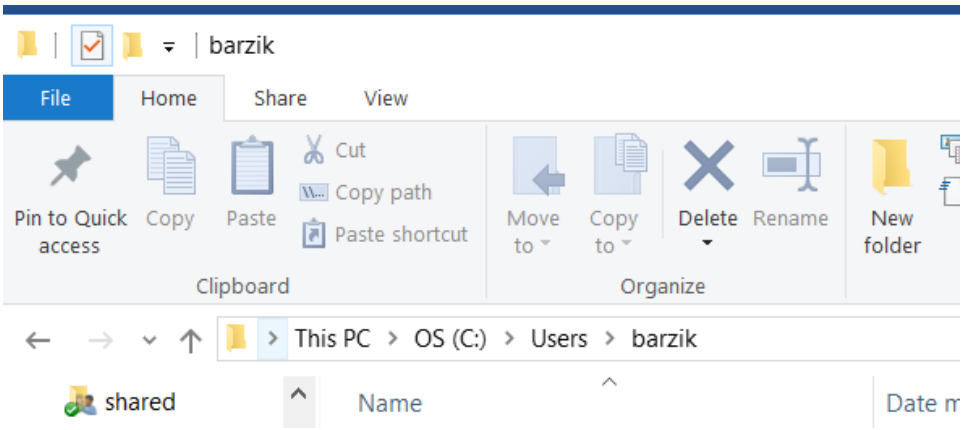
אנו יכולים לנווט שוב בחזרה באמצעות cd ושם התיקייה. אפשר גם להקליד ישירות את הנתיב:

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

C:\Users\barzik>cd ..

C:\Users>cd ..

C:\>cd Users\barzik\Documents

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

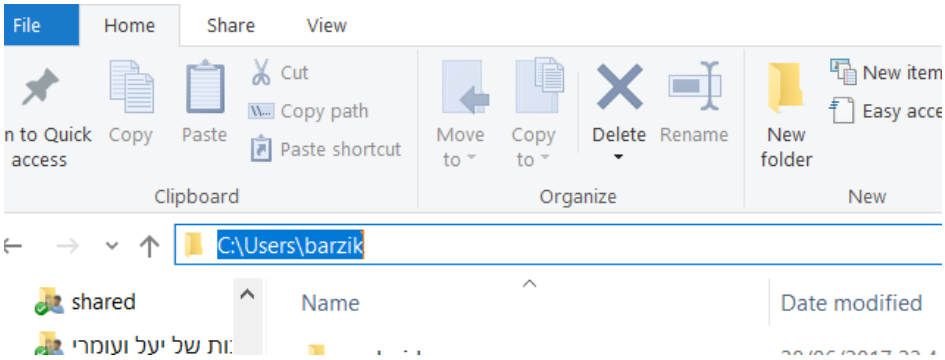
טיפ קטן – אפשר להקליד tab כדי לבצע השלמה.

מציאת מיקומים בטרמינל דרך חלונות

אם אתם רוצים להגיע לתיקייה מסוימת ולא בטוחים מה מיקומה, מצאו אותה בסייר הקבצים הגרפי ולחצו על הכותרת. למשל, התיקייה הזו:



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



אני יכול להעתיק את הנתיב אל הטרמינל. להקליד:

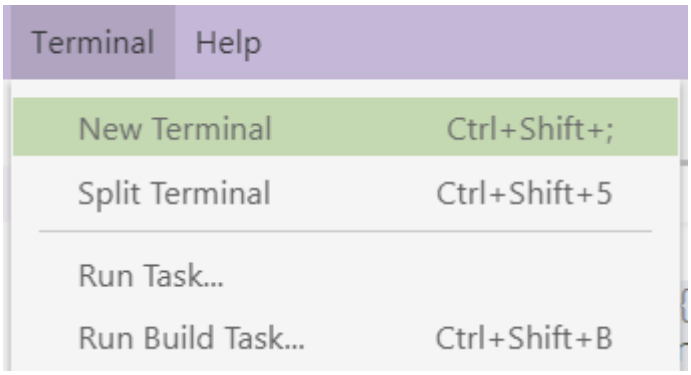
```
cd C:\Users\barzik
```

ואז ללחוץ על אנטר ולהגיע אל היעד המבוקש.

טרמינל ב-Visual Studio Code

אפשר לעבוד עם הטרמינל דרך ה-IDE החינמי Visual Studio Code, שמומלץ לכל מפתח ג'אווהסקריפט. העורך הזה, מבית מיקרוסופט, הוא חינמי, לא מכביד 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 למעלה ובחרו ב-New Terminal:



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

בדיקת גרסת Node.js דרך הטרמינל

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

```
node -v
```

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

אם הכול 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 node -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 תיקיית local תחת המשתמש שלי ונכנסתי אליה באמצעות:

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

לחלופין, פתחו את Visual Studio Code, צרו באמצעותו פרויקט במיקום מסוים ובאמצעות לחיצה על לשונית Terminal ו-New Terminal ייפתח לכם בתחתית 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 --template typescript
```

npx היא פקודת הפעלה של Node.js.
create-react-app הוא שם התוכנה שאנו מפעילים, Create React App.
my-app הוא שם האפליקציה שלנו. כשנפתח תוכנה אמיתית אנו נקרא לה מן הסתם בשם משמעותי יותר. כרגע נבחר ב-my-app – האפליקציה שלי.
--template typescript הוא הוראה ליצור את הפרויקט עם טייפסקריפט מובנה.

אם יש לכם Node.js במערכת והכול תקין, ההתקנה תעבור בקלות. היא נמשכת כמה דקות טובות, אין מה לחשוש. במהלך הדקות האלו התוכנה מורידה את כל המרכיבים מהרשת על מנת ליצור אצלכם 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 תיקייה שנקראת my-app. היכנסו אליה באמצעות:

```
cd my-app
```

והפעילו את המערכת החדשה שלכם באמצעות:

```
npm start
```

מדובר בפקודה של Node.js שפשוט מפעילה סקריפט שנקרא start. אם הכול תקין, הדבר הראשון שתשימו לב אליו הוא שנפתח לכם חלון של דפדפן שמכיל את האפליקציה שלכם! זה לא קסם. מה שתהליך ה-Node.js עושה הוא לייצר שרת על המחשב שלכם מאחורי הקלעים, לטעון את רכיבי הריאקט ואז לפתוח דפדפן שמכוון אל השרת המקומי שלכם יש מאין. אם תסתכלו על כתובת הדפדפן תוכלו לראות 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) תוכלו לראות שכבר יש מבנה בסיסי לאפליקציה. בתיקיית public אנו נראה שקובץ ה-HTML קיים והוא כבר מכיל:

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

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

```
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();
```

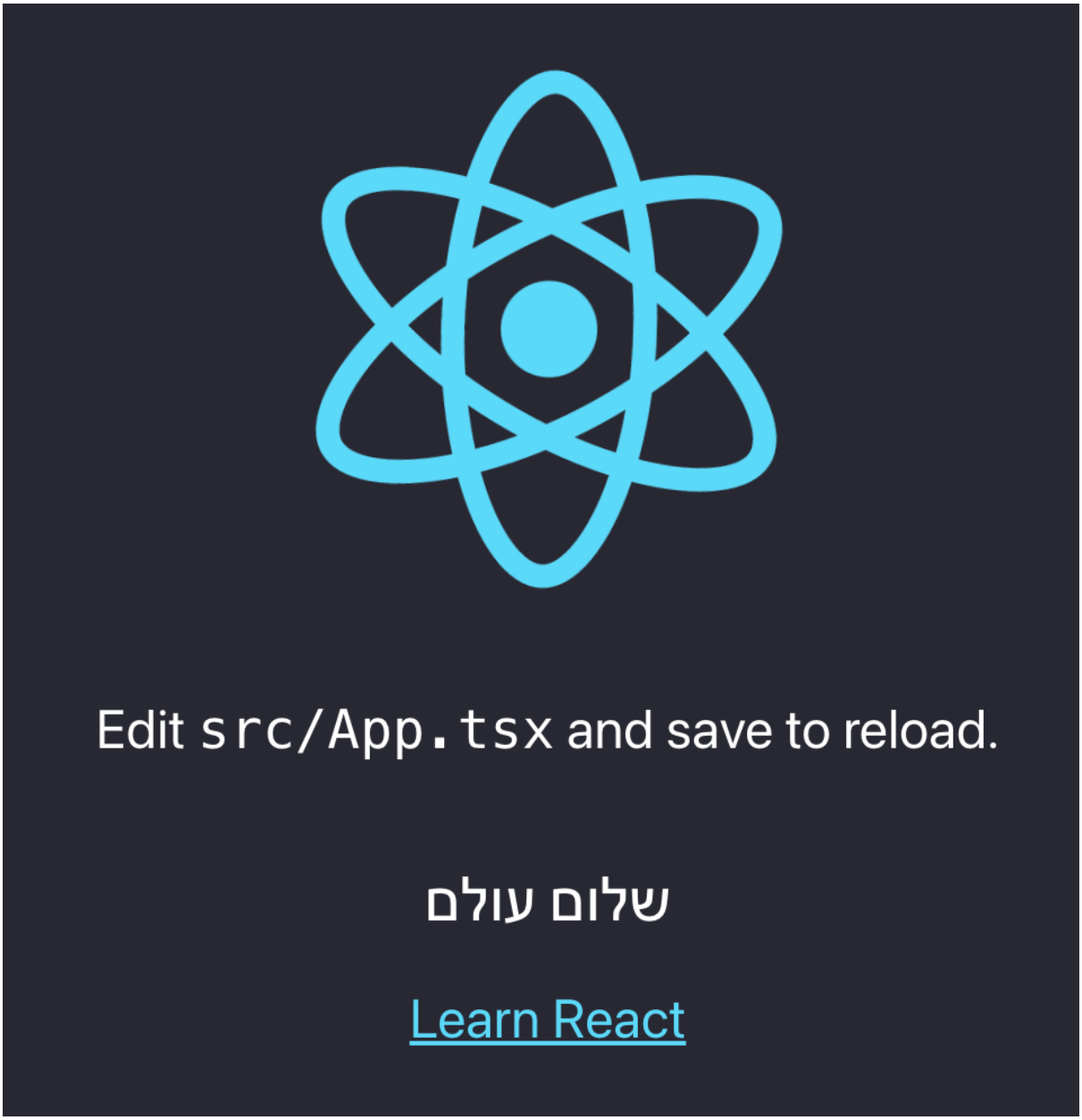
איזו קומפוננטה? App. איך מתבצעת ההפניה? באמצעות import, שנרחיב עליו בהמשך. באיזה קובץ נמצאת App? רואים את זה ב-import. בקובץ App.tsx:

```
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.tsx</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

אם נשנה משהו בקומפוננטה, למשל נוסיף שורה כמו:

```
<p>
שלום עולם
</p>
```

ונשמור, נוכל לראות שבמטה קסם, גם האפליקציה שלנו השתנתה ומופיע בה "שלום עולם".



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

שמות הסיומות של קבצים המכילים קומפוננטות של ריאקט הן tsx ולא ts. זו וריאציה של JSX רק עם טייפסקריפט. בפרויקטים שאין בהם טייפסקריפט, הסיומות הן jsx. אנו כמובן עובדים עם טייפסקריפט אז בכל קובץ שיש בו קומפוננטה של ריאקט או קוד ריאקטי אנו נשמור אותו בסיומת tsx. שם פורמט הנתונים הוא JSX, אך הסיומת היא tsx. אם יש לנו קבצים שיש בהם רק טייפסקריפט ללא קומפוננטות של ריאקט, הסיומת שלהם תהיה ts ולא tsx. אנו נרחיב על כך בהמשך, אבל כרגע כלל האצבע הוא סיומת tsx לקומפוננטות של ריאקט, סיומת ts לקבצים שבהם אין קומפוננטות כאלו.

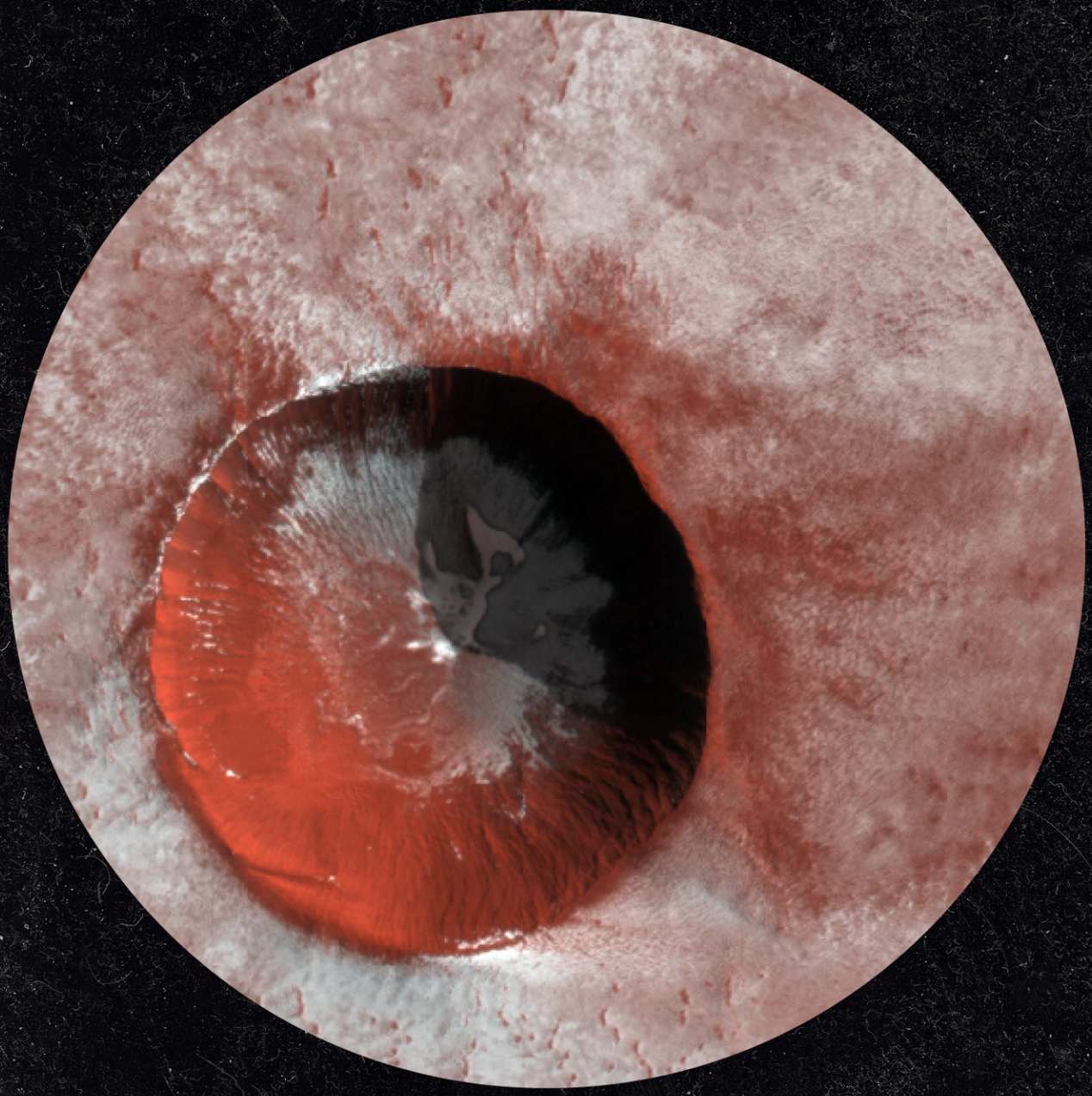
קשיים ותקלות

אם נתקלתם בקשיים בתהליך ההתקנה של Create React App, אל דאגה! יש לפעמים בעיות שנובעות משינויים במערכת ההפעלה, אנטי-וירוס שפתאום מתערב או בעיה אחרת שנובעת מכך שמערכות ההפעלה שלנו שונות. אבל כדאי מאוד לא לוותר. אם יש הודעת שגיאה – פשוט חפשו אותה בגוגל או בידקו עם בינה מלאכותית כמו צ'אט GPT. Create React App היא כל כך פופולרית, עד שיש תיעוד רב מאוד לגביה ברשת וגם תיעוד של כל תקלה אחרת. קיבלתם שגיאה אדומה ומפחידה? הדביקו אותה בחלונית השאלה של צ'אט GPT עם השאלה: "How to solve", חפשו אותה בגוגל, התייעצו לגביה בקבוצת פייסבוק או באתר stackoverflow. אל תדלגו על בניית סביבת העבודה הזו כי היא הבסיס העיקרי לעבודה עם ריאקט.

oliver@webiks.com

פרק 4

כתיבת קומפוננטה ראשונה ב-CREATE REACT APP



כתיבת קומפוננטה ראשונה ב-React App Create

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

הבה ניצור קומפוננטה בשם Greeting שבה יהיה כתוב "שלום עולם". ראשית – הקובץ: ניצור Greeting.tsx תחת תיקיית src. שימו לב ששם הקובץ מתחיל ב-G גדולה.
הקומפוננטה הזו אינה שונה כלל מקומפוננטה שלמדנו לעבוד איתה בסביבת העבודה הקודמת. היא קומפוננטה פונקציונלית שנראית כך:

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

סוג המידע הבסיסי שקומפוננטה ריאקט מחזירה הוא JSX.Element ואנו נגדיר אותו כך.

כיוון שנקרא לקובץ הזה באמצעות import, אנו צריכים לייצא את הפונקציה בדרך מסוימת. איך מייצאים? באמצעות סינטקס שנקרא export default ומגיע לפני הפונקציה שמייצאים. אנו נרחיב על import\export בהמשך, אבל גם בלי להבין לעומק זה הגיוני – אנחנו שמים את הקומפוננטה בקובץ נפרד – ולקובץ הזה אנו עושים import. אנחנו חייבים לעשות export בצד השני. את ה-export עושים כך:

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

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

בקובץ App.tsx נעשה import לקומפוננטה שלנו באמצעות:

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

מהרגע שעשינו import, אנחנו יכולים להשתמש ב-Greeting איפה שאנחנו 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.tsx</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 './Greetings';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</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 בדיוק כמו הדוגמה הפשוטה. ברגע שיש לי קומפוננטה אחת, אני יכול להשתמש בה כמה פעמים. זה שהיא בתוך קובץ ושאנחנו משתמשים ב-import\export מסתורי לא אמור להפריע לנו. בהמשך נלמד עוד על import\export.

מהרגע שנשמור גם את Greeting.ts וגם את App.ts – נראה מייד את התוצאה על האפליקציה בלי שיהיה צורך ללחוץ על F5 ולרפרש:

הקפיצה מסביבה לימודית פשוטה לסביבת פיתוח מורכבת נראית קשה ומפחידה, אבל ככל שתתאמנו יותר – כך תתרגלו אליה. יש בה המון דברים טובים – ראשית יש לנו את babel שמגיע בחינם, יש לנו שרת פיתוח, בדיקות (בהמשך נלמד על הבדיקות) וגם בודק תקינות קוד של eslint, שמתריע על בעיות שימנעו מהאפליקציה שלנו לרוץ. והכול באמת בא מהקופסה. אז גם אם זה נראה מורכב, נא לא להיבהל ולתרגל על הסביבה הזו. זו הסביבה שבסופו של יום תפגשו במקומות העבודה.

הערה בנוגע לריאקט מגרסה 16 ומטה:

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

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

השגיאה הזו מרמזת על כך שמשהו חסר. מה חסר? במקרה הזה הריאקט עצמו. בעוד בסביבת הלימוד הפשוטה שעליה למדנו, ריאקט נמצאה בכל מקום כי הכול היה בקובץ אחד, בריאקט מגרסה 16 אנחנו חייבים לכלול בכל קומפוננטה וקומפוננטה את כל הרכיבים שאנו צריכים להפעלתה. במקרה הזה: ריאקט. אנו חייבים "לייבא" את ריאקט ועושים את זה באופן הבא:

```
import React from 'react';

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

אז יש לנו כאן ייבוא של ריאקט, שהוא באמת פשוט וחיוני בכל קומפוננטה, יש לנו ייצוא, שיהיה חשוב כשנרצה להשתמש בקומפוננטה, ויש לנו את הקומפוננטה שלנו, שבמקרה הזה לא עושה המון אלא רק מחזירה טקסט. והכול נמצא בקובץ בודד אחד שנקרא Greeting.tsx. זה אחד מהשינויים המרכזיים בין ריאקט 16 לריאקט מגרסאות חדשות יותר. אם אתם לומדים מהספר הזה ומשתמשים בגרסה העדכנית של ריאקט, לא תקבלו שגיאה כזו. אך אם אתם עובדים על ראקט בסביבה אחרת, ישנה יותר, תהיו עירניים לעניין זה.

תרגיל:

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

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

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

פתרון:

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

```
Welcome.tsx
Inigo.tsx
Greeting.tsx
```

הקומפוננטות Inigo.tsx ו-Greeting.tsx הן קומפוננטות באמת פשוטות. הדבר היחיד שאנו צריכים לזכור זה את import React from 'react' ואת export default. כך הן נראות:

Inigo.tsx

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

Greeting.tsx

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

כל אחת

איך אני משתמש בהן? מבצע להן import מתוך Welcome.tsx ומשתמש בהן ב-JSX כרגיל. בדיוק כפי שהשתמשתי בסביבת הפיתוח הפשוטה. כל מה שאני צריך לזכור הוא לבצע import. זה הכול.

כך Welcome.tsx תיראה:

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

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

שימו לב שאני מבצע export default לקומפוננטת Welcome.

נותר לי רק לייבא את הקומפוננטה Welcome למקום שאני רוצה להשתמש בה, ב-App.tsx:

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

function App():JSX.element {
  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 לזו של Create React App הוא ה-import\export. בעצם כל קומפוננטה עושה import לדברים שהיא צריכה ו-export לעצמה, כדי שאחרות יוכלו להשתמש בה.

offer@webit.com

פרק 5

EXPORT / IMPORT

Import\Export

מדובר בחלק אינהרנטי מג'אווהסקריפט (וכמובן בטייפסקריפט), שחשוב לכל סקריפט בג'אווהסקריפט מורכבת. המנגנון הזה מאפשר לנו לעבוד עם כמה וכמה קבצים ולנהל את הקוד שלנו בקלות. בעבר היה מקובל לכתוב את כל הקוד בקובץ ארוך מאוד וזה כמובן קשה מאוד לניהול ולעבודה. כפי שראינו בסביבת הפיתוח המודרנית, כדאי לעבוד עם חלוקה של קבצים ועם import ו-export.

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

במקום שיש ייבוא חייב להיות ייצוא. הייצוא הבסיסי, הפשוט והקל להבנה מתקיים עם שתי מילים שמורות: export ו-default. המשמעות של export היא ייצוא והמשמעות של default היא ברירת מחדל. כשאנו כותבים שם של קלאס, פונקציה או משתנה ולפניו מציבים export default, אנו מייצאים את אותם קלאס, פונקציה או משתנה.

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

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

שימו לב שאין פה קלאס, אין פה משתנים – אני מייצא טקסט בלבד. כדי להשתמש בו, אני רק צריך לעשות import. ניכנס ל-App.tsx, נייבא אותו ונדפיס אותו בקונסולה. הקוד שלנו ייראה כך:

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

import './App.css';

import myVar from './Vars.tsx';


function App():JSX.element {

  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 שאני מגדיר את כל מה שמופיע אחרי ה-export default. במקרה הזה הגדרתי משתנה בשם myVar. אני יכול פשוט להדפיס אותו עם console.log. אסור לנו לשכוח שעם כל הכבוד לקומפוננטות, הן כתובות בטייפסקריפט שמומר לג'אווהסקריפט ומתנהגות כמו כל קוד אחר של טייפסקריפט. אם אני אשים console.log בפונקציה של הקומפוננטה אני אוכל לראות את המשתנה בקונסולה של הדפדפן. אם אני אפתח את כלי המפתחים, אני אכן אראה את הטקסט שעשיתי לו export:

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

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

אנחנו יכולים לייצא הכול. למשל, משתנה שיש בו טקסט או מספר:

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

export default someVar;
```

שימו לב שבשלב זה אני לא חייב שהשם של המשתנה יהיה זהה בייצוא ובייבוא. בדוגמה הזו אני מייצא משתנה שנקרא someVar, אבל מייבא אותו כ-myVar. אני יכול לקרוא לו בכל שם שאני רוצה כל עוד אני משתמש במילה השמורה default שעליה נלמד בהמשך.

אנחנו יכולים לעשות export לפונקציה. למשל:

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

כדאי לשים לב שמדובר בפונקציה רגילה ולא בפונקציית קומפוננטה (רואים את זה כי היא לא מתחילה באות גדולה ולא מחזירה JSX אלא מחרוזת טקסט). כשאני מייבא אותה (שוב, באיזה שם שאני רוצה) אני צריך להפעיל אותה בדרך כלשהי. בדוגמה הזו אני מבצע ייבוא בשורה הרביעית ומשתמש בפונקציה הזו:

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

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

export default App;
```

ואם זה נראה לכם מוכר, זו בדיוק הדרך לייבא ולייצא פונקציה שהיא קומפוננטה בריאקט. קומפוננטה בריאקט היא בסופו של דבר פונקציה ג'אווהסקריפטית לכל דבר, רק כזו שמשתמשת ב-JSX. אנחנו מייבאים אותה ומשתמשים בה ב-JSX (בניגוד לפונקציה רגילה, שלה אנו קוראים). זה הכול. כדאי לשים לב שסיומת שם הקובץ במקרה של פונקציות שאין בהן קוד JSX יהיה ts ולא tsx.

ייבוא מנתיבים אחרים

עד עכשיו ייבאנו קוד שנמצא בדיוק באותה תיקייה של הקומפוננטה שבה השתמשנו בקוד המיובא. אבל מה קורה אם הקוד שלנו נמצא בתיקייה אחרת? במקרה הזה אנו צריכים להקליד את הנתיב היחסי של התיקייה שלנו ביחס לקובץ שבו אנו נמצאים.

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

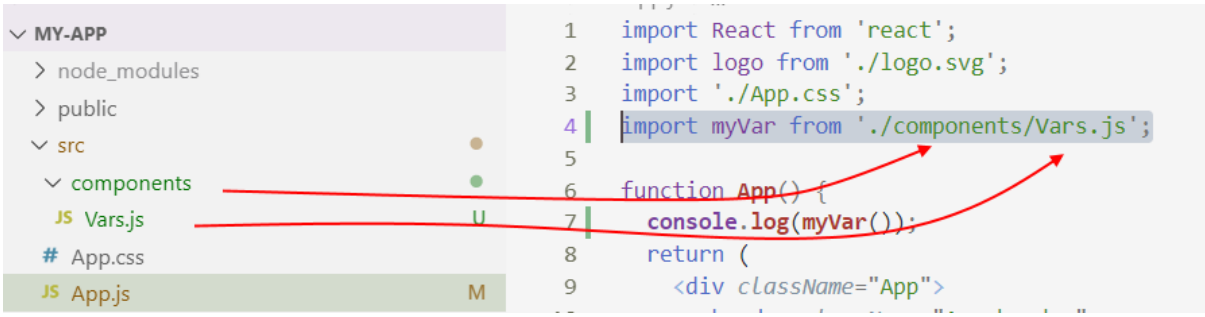
```
import myVar from './components/Vars.ts';
```

ממה הכתובת הזו מורכבת?

החלק הראשון - ./ - מסמן את המיקום של התיקייה שלנו, הקובץ שבו אני כותב את הקוד. מהנקודה הזו הכול מתחיל כיוון שאנו צריכים לכתוב נתיב יחסי מהמקום שבו אנו נמצאים.

החלק השני – תת-התיקייה components.

החלק השלישי – שם הקובץ כרגיל. אם מדובר בסיומת של ts היא לא הכרחית:



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

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

העניינים מסתבכים כאשר התיקייה שאנו רוצים לייבא ממנה היא לא תת-תיקייה של התיקייה הנוכחית שאנו נמצאים בה, אלא תיקייה שנמצאת מעל או תיקייה אחות. במקרה הזה אנו נעזרים ב-../ שאומר "לעלות תיקייה אחת למעלה".

למשל, הבה נניח שיש לאפליקציה שלנו מבנה תיקיות כזה:

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

אני רוצה ש-App.tsx תייבא מ-MyVar.ts. איך אני עושה את זה? אני צריך "לעלות" תיקייה אחת למעלה מ-application אל src ואז לרדת אל תיקיית components. את המסע הזה אני מתאר באמצעות import כך:

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

הבה נראה את המסע:

חלק א' – **./** אנו מתחילים מהמיקום שלנו.

חלק ב' – **../** עולים תיקייה אחת ל-src.

חלק ג' – יורדים אל תיקיית components.

חלק ד' – קובץ MyVar.ts.

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

אין חובה להשתמש בסיומת הקובץ ts

כברירת מחדל, אין צורך להשתמש בסיומת ts של הקובץ. אפשר לעשות את הייבוא באופן הבא:

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

ייצוא של כמה משתנים

בריאקט יש לנו כלל של קומפוננטה אחת בקובץ אחד, אבל לפעמים יש צורך בייצוא של משתנים רבים באותו קובץ, למשל משתנים. אנחנו לא נשים משתנה אחד בכל קובץ. במקרה הזה ניפרד לשלום מ-default ונייצא ישירות את מה שאנחנו צריכים לייצא. למשל:

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

הפעם, כיוון שאין לנו default, אנו חייבים לייבא את מה שאנו רוצים בדיוק בשם שבו אנו מייצאים אותו. איך? כך:

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

או:

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

אנחנו יכולים לייבא הכול כאובייקט ואז לקרוא למשתנים השונים כתכונות של האובייקט באמצעות ייבוא של הכול – שימוש בסימן כוכבית (*) ובמילה השמורה as. כשאני משתמש ב-import * as אני נדרש לציין את תכונות האובייקט שאני רוצה להביא. לשם הדוגמה במקרה שלנו:

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

והייבוא ילך 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 לטעינה של המשאבים

ב-Create React App שעליה אנו לומדים וכן בחלק מאפליקציות הווב המודרניות הוא הספרייה ווובפאק (webpack). הספרייה הזו היא מודול מבוסס Node.js ואחראית על תהליך הבילד (build) של האפליקציה, תהליך שגם במסגרתו מתנהלת טעינת הקבצים.

אנו מסוגלים להשתמש ב-import גם כדי לייבא תמונות וקובצי CSS. אפשר לראות ב-App.tsx כיצד מייבאים תמונה וגם CSS:

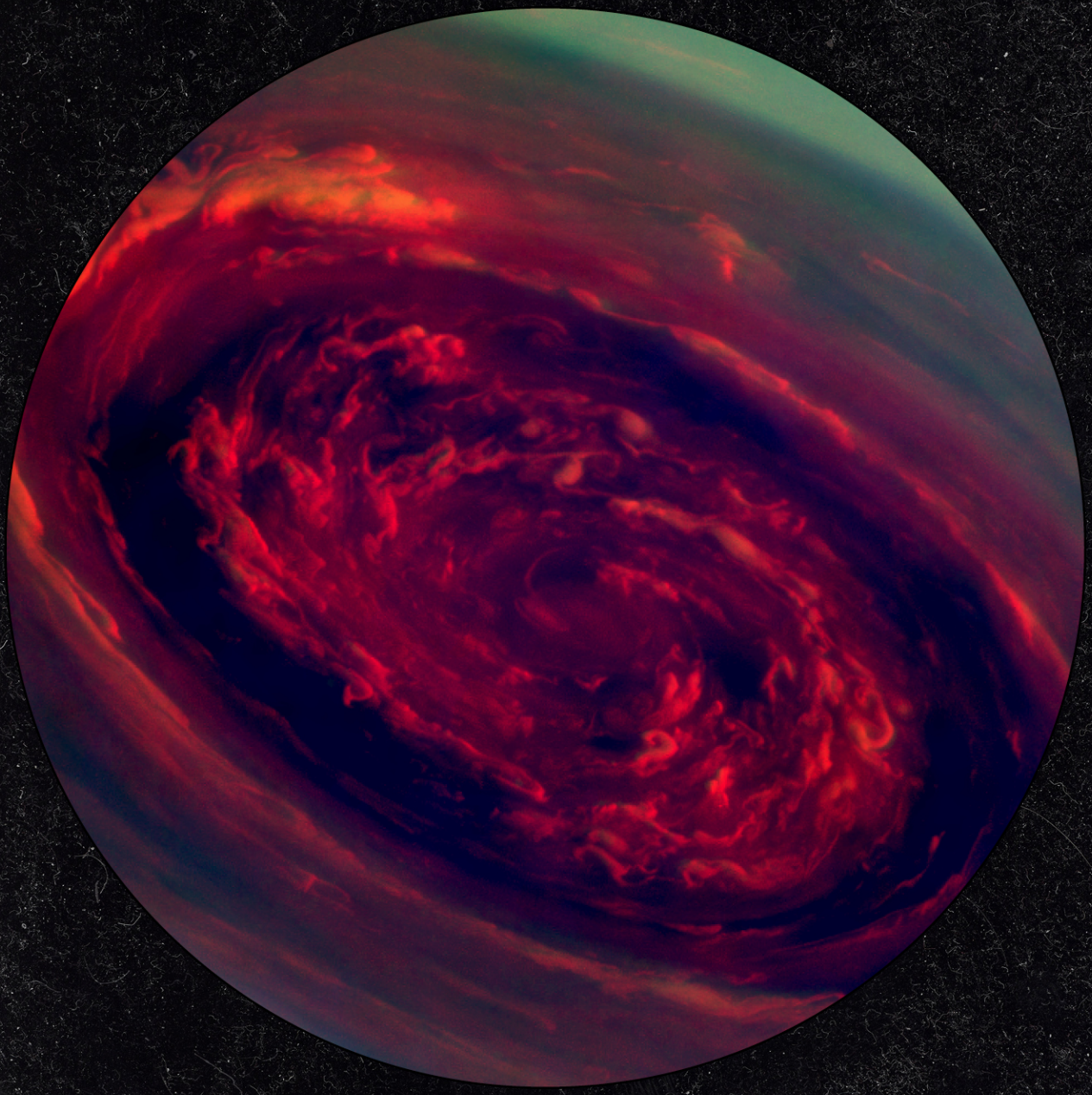
```
import logo from './logo.svg';
import './App.css';

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

export default App;
```

אני מייבא CSS וברגע שאני עושה את זה הוא מופעל ודואג לעיצוב ה-JSX/תמונות (ועל כך ארחיב בפרק הבא). את הייבוא והייצוא נתרגל בפרקים אחרים.

offworldwebtv.com

פרק 6

JSX



JSX

השימוש בטייפסקריפט והסיומת tsx עלולה לבלבל אותנו. המונח שמשמש אותנו בכל הנוגע לקומפוננטות הוא JSX. ג'אווהסקריפט שרץ בריאקט. אנו משתמשים בטייפסקריפט לשם פירוט סוגי המידע, אבל הקוד מומר ל-JSX ואז לג'אווהסקריפט רגיל.

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

הכלל הוא פשוט – כל ביטוי שמופיע בין סוגריים מסולסלים { } ונמצא ב-JSX בעצם נחשב לג'אווהסקריפט ורץ כג'אווהסקריפט והתוצאה מופיעה בקומפוננטה. כלומר, אם יש לי משתנים ואני רוצה להדפיס אותם,אני אכניס אותם ל-JSX עם סוגריים מסולסלים:

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

function App():JSX.Element {
  const myVar:string = '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 הזו אני מכריז על משתנה myVar ומעביר אותו ל-TSX. הקומפוננטה כבר תדאג להציג אותו. אם אציג את הקומפוננטה באמצעות מבט בדפדפן (בכתובת localhost:3000) אני אראה את הטקסט של המשתנה.
מובן שאני יכול להציב סוגריים מסולסלים כמה פעמים שאני רוצה. למשל:

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

function App():JSX.Element {
  const myVar:string = 'Hello World!';
  const anotherVar:string = '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 הזה ידפיס את המשתנים במקום המתאים.

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

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

function App():JSX.Element {
  const myVar:JSX.Element = <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 שנכנס לתוך משתנה משתנים בתוך סוגריים מסולסלים.
העניין הוא ש-TSX הוא ג'אווהסקריפט טהורה ויש להתייחס אליו בהתאם. אפשר לעשות איתו הכול בדיוק כמו בג'אווהסקריפט. למשל, להכניס בתוכו ביטוי:

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

אם תציבו את ה-JSX הזה בקומפוננטה שלכם, תראו שהוא מדפיס 2.

אבל הכוח של JSX הוא לא בתעלולים נלוזים כאלו אלא במקומות אחרים. אפשר להשתמש בו בלולאות. למשל:

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

function App():JSX.Element {
  const myChildren:string[] = ['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:string[]  = ['Omri', 'Kfir', 'Daniel', 'Michal'];
```

ניתן לראות שכנדרש בטייפסקריפט הגדרתי את הסוג של המערך כמערך של מחרוזות טקסט. נכון, SX נראה קצת זר ומוזר אבל הוא טייפסקריפט והיכן שיש לי משתנים, קלט או פלט, אני חייב להגדיר את הסוג שלהם כטייפסקריפט.

החלק השני הוא הגדרת סוגריים מסולסלים בתוך ה-return כדי שנוכל לעבוד עם ביטויי ג'אווהסקריפט.

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

בתוך הסוגריים האלו אנו יכולים לכתוב כל ביטוי ג'אווהסקריפטי שהוא. במקרה הזה אנו נעשה map פשוט למערך שיחזיר לנו בכל פעם שם של ילד אחר.

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

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

אני יכול להשתמש בו במשפט תנאי. למשל:

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

function App():JSX.Element {
  const someVar:boolean = 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 logo from './logo.svg';
import './App.css';

function App():JSX.Element {
  const myChildren:JSX.Element[] = [<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 יצרתי מערך של אלמנטים. כדי להדפיס אותו אני לא חייב להשתמש ב-map אלא רק לשים אותו ב-JSX והוא יודפס אוטומטית. זה לא יעבוד עם אובייקט, רק עם מערך. חשוב לשים לב שהמערך הוא מערך של אלמנטים של JSX ולפיכך אנו מגדירים את הסוג שלו.

בעתיד, כשנראה איך המשתנים האלו מגיעים מצד השרת או מקלט של המשתמש, הכול ייעשה הרבה יותר מעניין, אבל כבר עכשיו זה אמור לטלטל את עולמכם. עד כה אמרו לנו לא לערבב HTML וג'אווהסקריפט. אבל חשוב לזכור שזה אולי נראה כמו HTML, אבל זה לא HTML. JSX שונה מ-HTML כי הוא ג'אווהסקריפט. כך, למשל, אם נכתוב את המילה class, שהיא ולידית לחלוטין ב-HTML אך מילה שמורה בג'אווהסקריפט, נקבל שגיאה.

לדוגמה, הקוד הזה:

```
function App():JSX.Element {
  const myChildren:JSX.Element[] = [<li>Omri</li>, <li>Kfir</li>,
<li>Daniel</li>, <li>Michal</li>]
  return (
    <div class="App">
    <header class="App-header">
      <img src={logo} class="App-logo" alt="logo" />
      <ul class="My-children" >
        {myChildren}
      </ul>
    </header>
    </div>
  );
}
```

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

```
function App():JSX.Element {
  return (
    <img src="./src/header.png">
  );
}
```

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

```
function App():JSX.Element {
  return (
    <img src="./src/header.png" />
  );
}
```

ההבדל נראה קטן, אך הוא משמעותי. לא מדובר פה ב-HTML אלא ב-JSX, ג'אווהסקריפט שעובד עם אובייקטי XML, שאלו גם האובייקטים ש-HTML עובד איתם.

רשימות ב-JSX

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

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

איך אני אדפיס 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 הזה – המיקום של הפריט ברשימה, אם נרצה להציג את המערך כמו שצריך ברשימה הזו:

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

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

מה שחשוב פה הוא ה-key שמקבל את ה-index – כלומר המיקום של המשתנה באיבר.

ההנחה המובלעת היא שהקוראים מכירים את פונקציית map ואת פונקציית חץ.

אם הקוד הזה:

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

נראה לכם לא מובן, כדאי ללמוד שוב על פונקצית map ועל פונקציית חץ, משום שהן קריטיות להבנה ומשתמשים בהן המון בריאקט.

תרגיל:

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

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

פתרון:

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

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

export default TodayTime;
```

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

את הקומפוננטה אני צורך (כלומר מייבא ומשתמש בה) כרגיל איפה שאני רוצה. במקרה הזה, ב-App:

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

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

export default App;
```

תרגיל:

הדפיסו, 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:

הדפסת לולאת for מ-10 עד 0 עובדת כך:

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


  }
```

פתרון:

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

function App():JSX.Element {
  const items:JSX.Element[] = [];
  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 ומאכלסים אותו באמצעות לולאת for פשוטה. בכל איטרציה אנו משתמשים במתודת push שמכניסה איבר חדש למערך. מה היא מכניסה? JSX של אלמנט <li> עם i. כדאי לשים לב שכיוון שאנו רוצים שה-i יחזיר את ערכו – אנו שמים אותו בסוגריים המסולסלים. לא נשכח גם לשים index ב-key.

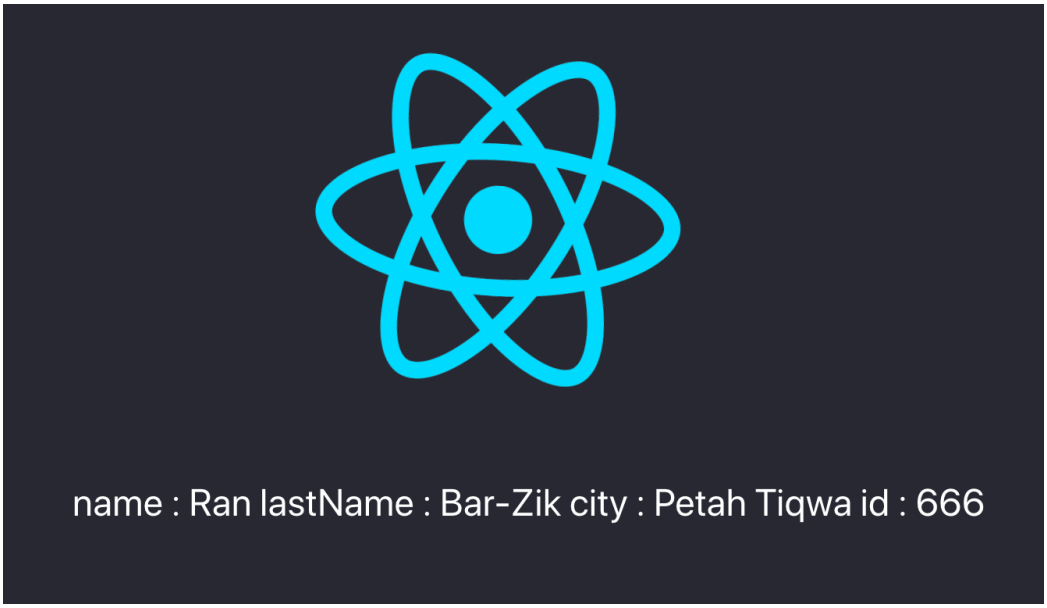
כל מה שנותר לנו לעשות הוא לקחת את המערך ולהדפיס אותו. כאמור, JSX יודע לקחת מערך ולטפל בו אוטומטית.

תרגיל:

נתון אובייקט משתמש:

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

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

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



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

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

פתרון:

```
import logo from './logo.svg';
import './App.css';
function App():JSX.Element {
  type UserInfo = {
    name: string,
    lastName: string,
    city: string,
    id: string,
  }

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

ראשית, הדבקתי את האובייקט שנדרש ממני כדי לפתוח בתרגיל. שנית, יצרתי פונקציית map ואת ה-JSX והכנסתי אותו לתוך מערך באמצעות push. השלב הבא הוא פשוט לשים את המערך הזה בתוך ה-JSX ולסמוך עליו שירנדר הכול.

oliver@webiks.com

פרק 7

קומפוננטה עם תכונות (PROPS)

קומפוננטה עם תכונות (props)

הקומפוננטות שבנינו 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 בפיתוח מונחה קומפוננטה ובדרך כלל נמצא קומפוננטות ללא קלט רק בתחילת שרשרת הקומפוננטות שלנו.

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

ראשית, ניצור את הקומפוננטה הבסיסית בקובץ Welcome.tsx.

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

export default Welcome;
```

עכשיו נוסיף את הטקסט שלנו:

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

export default Welcome;
```

ואם אני ארצה לצרוך את הקומפוננטה הזו, אני אצרוך אותה עם import (שימו לב שאולי אצטרך לשנות את הנתיב, כפי שלמדנו בפרק על import ו-export):

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

ואשתמש בה בכל מקום פה:

```
<Welcome />
```

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

```
function Welcome(props)
```

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

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

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

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

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

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

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

ה-href וה-title נקראים "תכונות", וזו הדרך שלנו לתקשר עם רכיבי HTML ולשנות אותם. אותו הדבר עם קומפוננטות ריאקטיות. באמצעות שימוש בתכונות אנו מעבירים להן מידע.
אנו נגדיר גם את סוג המידע של התכונות עם טייפ כמקובל בטייפסקריפט. אם שכחתם איך יוצרים type ומה הוא אומר – זה הזמן לבצע חזרה באמצעות החלק על טייפסקריפט בספר "ללמוד ג'אווהסקריפט בעברית". בו יש הסבר מפורט על סוגי מידע. אנו חייבים להגדיר סוג מידע שאנו מצפים לקבל בקומפוננטה. אם לא, Create React App יסרב לעבוד. הגדרת סוג מידע חשובה כיוון שהיא מונעת תקלות ובעיות בהעברת מידע לקומפוננטות שאחרים כותבים.

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 כזה דבר? בקומפוננטה שלי אני אגדיר את הדרך שבה שולחים לי את התואר, למשל prefix וכמובן את סוג המידע, שהוא מחרוזת טקסט:

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

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

export default Welcome;
```

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

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

ומה אראה?

```
Hello, Doctor Moshe
```

הכוח האמיתי של ה-props הוא לא במשלוח מחרוזות טקסט. אני יכול לשלוח הכול. ב-JSX אני יכול לשלוח למשל אובייקטים שלמים. בואו נניח שיש לי אובייקט משתמש עם שם ותואר, משהו כזה:

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

איך אני שולח את האובייקט הזה?

אני יכול לעשות משהו כזה:

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

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

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

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

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

export default Welcome;
```

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

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

function App(): JSX.Element {

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

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

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

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

export default Welcome;
```

מה קורה פה? פשוט מאוד. האובייקט של props.user מורכב כזכור גם מ-prefix וגם מ-name.

במקום לעשות משהו כזה:

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

אני עושה משהו כזה:

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

זה בדיוק 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 מרִנדור ומה שאנו רואים. בהמשך נלמד על כך שהפלט האמיתי הוא האירועים שמופעלים בקומפוננטה.

תרגיל:

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

```
Happy Birthday! You are X years old!
```

כש-X הוא הגיל. קראו לקומפוננטה הזו מ-app.

פתרון:

הקומפוננטה נשמרת בשם Birthday.tsx והיא מכילה את הקוד הזה:

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

ב-app היא תיקרא כך:

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

אפשר לראות שהדבר היחיד שמעניין כאן הוא שיש לי props ומהם אני לוקח את ה-age. כשאני קורא לקומפוננטה הזו, אני מעביר לה age, בדיוק כפי שלמדנו. אני גם מגדיר את סוג המידע של הגיל כמספר, כמקובל בטייפסקריפט.

תרגיל:

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

```
Happy Birthday Y! You are X years old!
```

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

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

פתרון:

קומפוננטת Birthday:

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

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

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

הוא בדיוק כמו:

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

אבל השימוש ב- destructuring אלגנטי יותר.

תרגיל:

הארכיטקט הבכיר בחברה קובע שהקומפוננטה שלכם תקבל את האובייקט של ה-user, בסגנון הזה:

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

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

פתרון:

כיוון שכפו עלינו שינוי, אנו חייבים לעבוד עם האובייקט. אם באמת השתמשתם ב-destructuring, יהיה לכם הרבה יותר קל לעבוד. צריך רק להבין שבמקום props.name ו-props.age יש לנו props.user:

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

תרגיל:

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

פתרון:

אסור לנו לשכוח שאף על פי שב-JSX עסקינן, עדיין מדובר בטייפסקריפט פשוטה וקלה, שאנו מכירים. אם אתם מתכנתים, אתם אמורים לשלוט במשפטי תנאי או במשפטי תנאי מקוצרים. הדרך להכניס לקבוע phrase טקסט לפי הגיל היא באמצעות תנאי מקוצר:

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

או באמצעות משפט תנאי פשוט יותר אך אלגנטי פחות:

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

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

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

זה הכול. שוב, עם כל הכבוד ל-JSX ולסינטקס המבלבל – זה בסופו של דבר ג'אווהסקריפט.

oliver@webnik.com

פרק 8

דיבאג

דיבאג

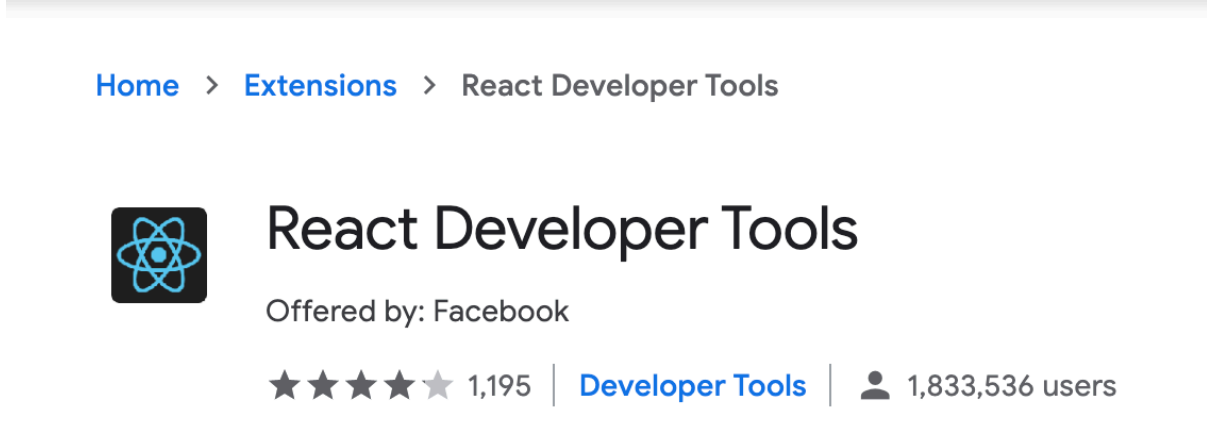
חלק משמעותי מכל פיתוח תוכנה הוא הליך הדיבאגינג – כלומר מציאת התקלות, השגיאות והבעיות שיש בקוד שאנו מריצים. דיבאגינג פירושו מציאת באגים, וכשאנו כותבים קוד תמיד, אבל תמיד, יהיו בעיות.

בדומה לג'אווהסקריפט, אנחנו יכולים לבצע דיבאג לכל קומפוננטת ריאקט עם פיירפוקס או כרום בקלות וביעילות כשאנו מריצים את סביבת הפיתוח שלנו. נוסף על כך, יש לכרום ולפיירפוקס תוסף של כלי מפתחים ייעודי לריאקט שאותו כדאי להכיר ולהתקין עכשיו. חפשו "React Developer Tools" בחנות התוספים של הדפדפן והורידו אותו או הקליקו על:

http://bit.ly/reactdevtool לכרום

או על:

https://addons.mozilla.org/en-US/firefox/addon/react-devtools/ לפיירפוקס.

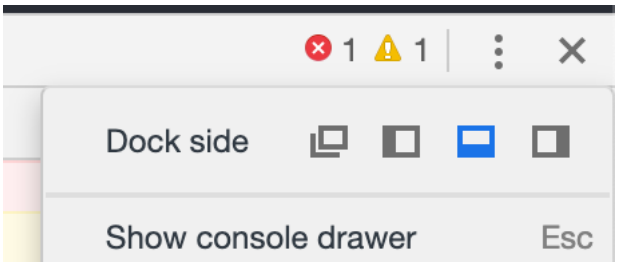


על מנת להדגים את תהליך הדיבאג, נחוץ לנו קוד לדבג. אתם יכולים להשתמש בכל קוד שכתבתם עד כה. בפרק הזה אני אדגים על הקומפוננטה שיצרנו בתרגיל בפרק הקודם. אבל התהליך עובד, כמובן, בכל קומפוננטה שהיא ובכל קוד שהוא. הדיבאגר המובנה בדפדפן והתוסף המיוחד שמתלווה אליו אמורים להיות ידידיכם הטובים ביותר.

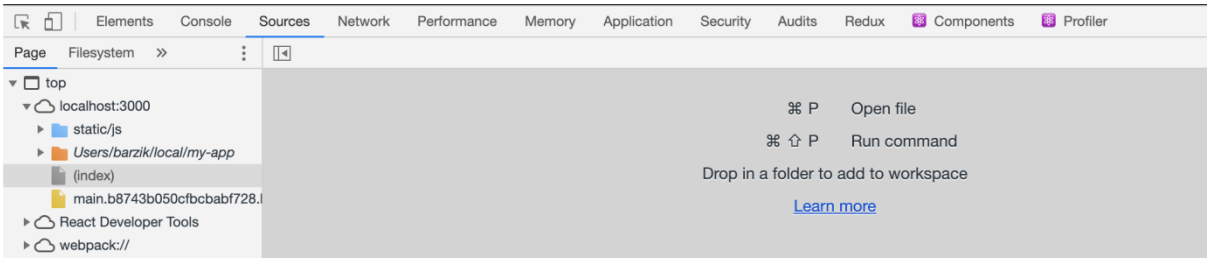
ראשית, הדיבאגר של הדפדפן עובד כרגיל. בכל דפדפן מודרני יש כלי מפתחים מובנה ובפרק זה נלמד על כרום, הדפדפן הנפוץ כיום. אך יש כלי מפתחים זהה בכל דפדפן עם אותו מבנה. בכרום אנו פותחים את כלי המפתחים באמצעות צירוף המקשים קונטרול, שיפט ו-i בחלונות או בלינוקס וקומנד, אופשן ו-i במק. ייפתח לכם כלי המפתחים בתחתית המסך.

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

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

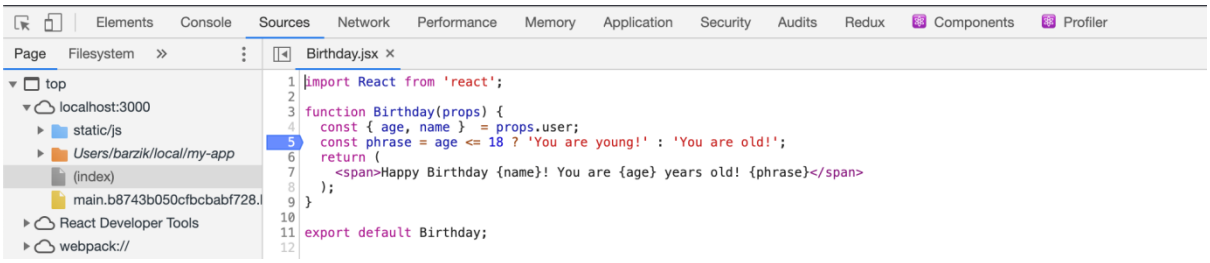


נעבור לתגית ה-sources:

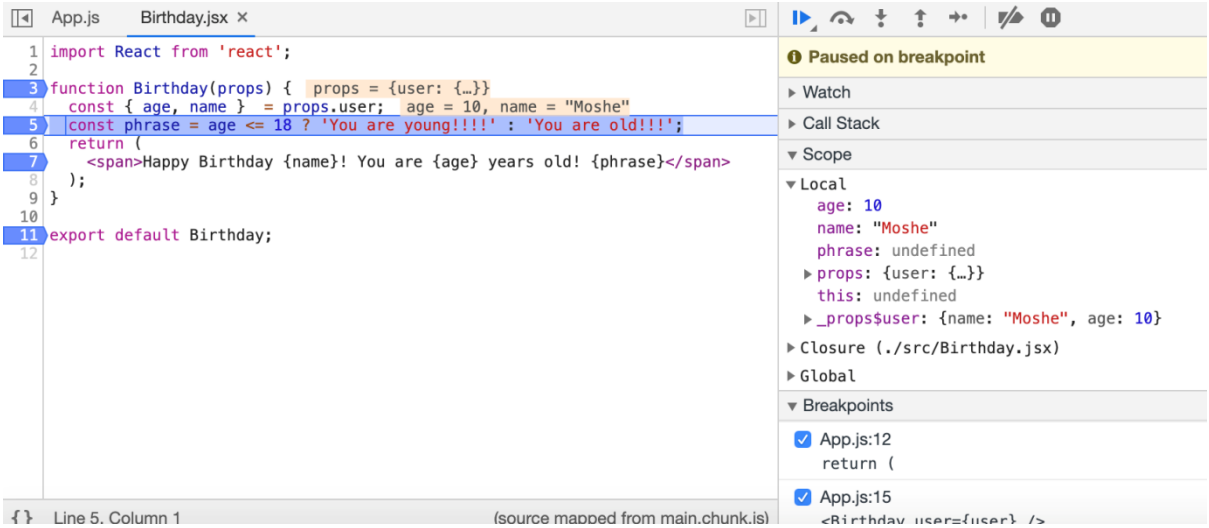


אנו יכולים לראות את מבנה האפליקציה שלנו והקומפוננטות בצד שמאל. אם אנו רוצים לחפש קובץ מסוים, לחיצה על קונטרול ו-p בחלונות ובלינוקס או קומנד ו-p במק תפתח תפריט חיפוש.

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

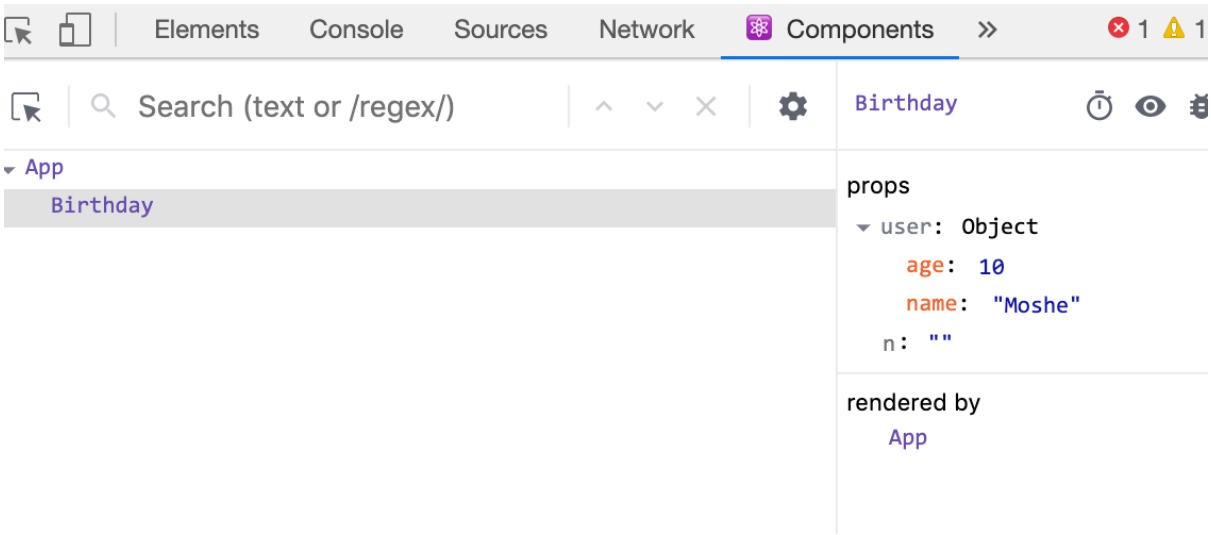


אנו יכולים, באמצעות לחיצה בצד השורה המתאימה, ליצור נקודת עצירה (Breakpoint). מדובר בנקודה שבה הסקריפט שלנו עוצר ולא ממשיך הלאה ואנו יכולים לראות מה מצב הקוד, לבדוק ולראות את ערכי המשתנים ולאחר מכן להמשיך את ההרצה. אם נעלה מחדש את העמוד (באמצעות F5 למשל או קונטרול + r) התוכנה תעצור בנקודת העצירה הזו ונוכל לראות את ערכי המשתנים, להקליד בקונסולה ולצעוד קדימה בזהירות כדי להבין מה השתבש:



אפשר גם ליצור נקודות עצירה באפליקציית הריאקט שנמצאת ב-app, בכל קובץ שהוא. חשוב לזכור שריאקט בסופו של דבר היא ג'אווהסקריפט. לא יותר, לא פחות.

אך התקנת תוסף הפיתוח של ריאקט פותחת לנו אפשרויות נוספות. החשובה ביותר היא צפייה ברשימת 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 ובאיזו אפליקציית ריאקט היא נמצאת:



לחיצה על אייקון העין תמקד אתכם באלמנט הקומפוננטה כפי שהוא נמצא ב-HTML, ולחיצה על האייקון של הבאג תציג את המידע של הקומפוננטה בקונסולה.

להרבה מפתחים מתחילים יש נטייה להימנע מהדיבאג כיוון שמדובר בכלי שנראה מפחיד וגם כך יש המון מה ללמוד – אבל זה מצער. כדאי להשקיע כמה שעות בניסיון להבין איך הוא עובד ולהשתמש בו במקום לנסות לפענח מה הבעיה בעצמכם או בעזרת כל מיני מעקפים כמו console.log. זה הרבה יותר קל ומהיר, ובטווח הארוך - משתלם מבחינת הזמן. כלי המפתחים משתכללים ומשתפרים כל הזמן, וכך גם התוסף של ריאקט. יש סיכוי שבזמן שתקראו את הפרק, התצוגה של התוסף או של כלי המפתחים תשתנה – הם עוברים שינויים כל הזמן. אבל הפונקציונליות הבסיסית של כלי המפתחים לא משתנה – מקום לשים נקודת עצירה, להריץ קדימה ולראות מה מצב ה-props. אלו הדברים החשובים לכם.
מבחינת תרגול – נוותר עליו בפרק הזה. התרגול שלכם יהיה בשימוש בדיבאגר בפרקים הבאים.

oliver@webnia.com

פרק 9

סטייט

סטייט

עד כה יצא לנו לעבוד עם קומפוננטות סטטיות בלבד, כלומר כאלו שיש בהן רק return של JSX או שמקבלות props ואז מחזירות JSX. קומפוננטות אלו הן מצוינות אבל "חסרות זיכרון". כלומר, אני מעביר לקומפוננטה מידע, היא מרנדרת אותו וזהו. גם אם אני אשנה את המידע, אחרי שהקומפוננטה הוצגה, אני עדיין אראה את המידע שהכנסתי בהתחלה.
אדגים את הבעיה עם קומפוננטה שנקראת CountUp.tsx. זוהי קומפוננטה שעושה משהו פשוט מאוד - היא מציגה כפתור ובכל פעם שאני לוחץ על הכפתור, הספרה עולה ב-1.
איך אני ממש דבר כזה בג'אווהסקריפט? בקלי קלות עם אירוע שצמוד לכפתור ויפעיל פונקציה כזו:

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

לא משהו מורכב מדי. שימו לב שכיוון שהפונקציה לא מחזירה דבר, סוג המידע שאני מפרט עבורה הוא void.

אני יכול להכניס את הקוד הזה בקומפוננטה באופן הבא:

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

שימו לב: אין בעיה להשתמש בפונקציה בתוך פונקציה. הפונקציה שהגדרנו בתוך הפונקציה תהיה זמינה אך ורק בתוך הסקופ של הפונקציה.

שימו לב: הפונקציה שאנו מפעילים ב-onClick מוקפת גם היא בסוגריים מסולסלים כיוון שאנו רוצים ששמה יתורגם לקוד. אנו נמצאים ב-JSX וצריכים להבהיר שמדובר בפונקציה של הקומפוננטה.

אם אני אקרא לקומפוננטה הזו ב-app באופן הבא:

```
import './App.css';
import CountButton from './CountButton';
function App():JSX.Element

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

export default App;
```

אני אראה שאכן הקומפוננטה מתרנדרת ומוצגת הספרה 0, אך הספרה לא מתקדמת בלחיצת הכפתור. אם תציצו בדיבאגר, שעליו למדנו בפרק הקודם, תוכלו לראות שאכן הפונקציה countUp נקראת בכל לחיצה, אבל המספר לא מתעדכן. מדוע? כדי להבין את זה לעומק אנו נדרשים להבין יותר מהו רְנדור.

ריאקט ורְנדור

כאשר אנו כותבים קומפוננטה ב-JSX של האפליקציה שלנו, אנחנו לא מייצרים HTML, אלא אובייקט ריאקטי שריאקט מכניסה ומציגה אותו ב-DOM האמיתי. פעולת ההצגה הזו, שמתרגמת את ה-JSX לתצוגת HTML שהדפדפן יודע לטפל בה, נקראת "רְנדור" מלשון render – הרצה. הרְנדור הראשוני של כל קומפוננטה מתבצע בטעינה הראשונית ואנו רואים את התוצאה שלו בדפדפן. כל עוד אין רְנדור מחדש, התצוגה בקומפוננטה לא תשתנה. זה בדיוק מה שקורה עם הקומפוננטה CountButton. הפונקציה שנקראת מהכפתור אכן עובדת ומעדכנת את המשתנה הפנימי, אבל כל עוד אין רְנדור מחודש, לא נראה את זה בדפדפן גם אם המשתנה הפנימי השתנה. על מנת להציג את השינויים שנעשו בקומפוננטה אנו חייבים לבצע רְנדור מחדש. הרְנדור מחדש נעשה באמצעות אחת משתי דרכים:

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

ברגע שיש רְנדור מחדש, הקומפוננטה מתעדכנת וגם ה-JSX, ואנו רואים את הערכים החדשים ב-JSX. בדוגמה שלנו, אם אנו רוצים שהקומפוננטה תציג את הערך של i שהשתנה, אנחנו צריכים לדאוג לשנות את הסטייט הפנימי שלה. בשביל זה אנחנו צריכים לדעת מה הוא סטייט.

סטייט

סטייט הוא בעצם הזיכרון הפנימי של הקומפוננטה. לקומפוננטות שעבדנו איתן עד כה לא היה זיכרון. או שהן קיבלו את המידע שלהן מבפנים (קומפוננטות תצוגה) או שהן קיבלו את המידע שלהן מקומפוננטת אב דרך ה-props. כך או אחרת, הן קיבלו את הקלט, עשו איתו חישוב והעבירו אותו ל-JSX. הקומפוננטה רונדרה, ה-JSX הוצג בתור HTML ובעצם הסיפור הסתיים. פה אנו זקוקים לזיכרון פנימי שיאפשר לקומפוננטה לנהל את המשתנים שלה, וחשוב מכך – לסמן לריאקט מתי בדיוק לרנדר את הקומפוננטה מחדש. הסטייט במקרה של ריאקט הוא אובייקט שאנו מכניסים אליו את המידע שאנו רוצים "לזכור". כאשר אנו משנים את המידע שיש בזיכרון, יש לנו רְנדור מחודש.

אז איך מנהלים את הסטייט? בקומפוננטות מסוג פונקציות אנו משתמשים במנגנון פשוט שנקרא הוקים (hooks) או ביחיד הוק. בהוק אנו מבצעים שתי פעולות:

1. הגדרת החלק שאנו מכניסים לסטייט.
2. הגדרת הפונקציה שבאמצעותה אנו משנים את החלק הזה בסטייט. בקריאה לפונקציה הזו מתבצע הרְנדור מחדש.

זה נעשה באמצעות ההוק useState. הוק הוא שם מפחיד לפונקציה פשוטה שמקבלת פרמטר אחד ויחיד – הערך ההתחלתי של הסטייט שלי. הפונקציה מחזירה מערך עם שני חלקים. החלק הראשון הוא הסטייט עצמו, שבו אני יכול להשתמש ב-JSX (או בכל מקום בקומפוננטה), והוא לקריאה בלבד. החלק השני הוא הפונקציה שבאמצעותה אני משנה את הסטייט.
בטייפסקריפט אני גם נדרש להגדיר את סוג המידע של הסטייט. useState בטייפסקריפט מאפשרת לי באמצעות גנריות להעביר את סוג המידע הזה. להזכירכם, בטייפסקריפט יש פונקציות התומכות בהעברת סוג המידע שהן אמורות להחזיר וזה נעשה עם חיצים משולשים והגדרת סוג המידע.

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

```
import { useState } from 'react';
function CountButton():JSX.Element {
  const [count, setCount] = useState<number>(0);
  function countUp():void {
    setCount(count + 1);
  }
  return (
    <div>
      <h2>{count}</h2>
      <button onClick={countUp}>Clck me</button>
    </div>
  );
}
export default CountButton;
```

ראשית, אני קורא להוקים באמצעות import. ה-import הזה משתמש ב- destructuring שלמדנו עליו בפרקים קודמים. קבלו אותו כמות שהוא - בכלים האלה צריך להשתמש כאשר אנו נעזרים בהוקים.

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

למשל, אם המשתנה שרציתי להכניס לזיכרון הוא count, שם הפונקציה שמשנה את המשתנה בזיכרון של הקומפוננטה יהיה countYp. במקרה שלנו שם המשתנה הוא: count, אז שם הפונקציה שמשנה את המשתנה בזיכרון יהיה setCount. זה הכול.
אני גם צריך להגדיר את סוג המידע של המשתנה שיש ב-state. אני עושה את זה עם גנריות שהסינטקס שלה הוא סוגריים משולשים וסוג המידע שאני מצפה לקבל מ useState. במקרה הזה מספר.
אני מבצע את ההגדרה הזו באמצעות השורה:

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

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

עכשיו אני צריך לשנות בקומפוננטה ולהמיר את כל הפעמים שבהן אני משנה את הערך של i בפונקציה של setCount, הפונקציה שאומרת בעצם – תכניס לזיכרון את הערך הזה, במקרה שלנו – count + 1:

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

זה הכול!
הבה נסקור את הפעולות שעשינו כדי ליצור "זיכרון" לקומפוננטה:

שם הפעולה	הקוד הישן	הקוד החדש
להביא את ההוקים	ללא	`Import { useState } from 'react;'`
לקבוע ערך ראשוני לסטייט ולקבל את המשתנה לזיכרון, לקבל פונקציה שמשנה אותו	`let count = 0;`	`const [count, setCount] = useState(0);`
להגדיר ל-useState את סוג המידע שאנו מוכנים שהיא תקבל	`let count:number = 0;`	`useState<number>(0);`
לשנות את הערך	`count = count + 1;`	`setCount(count + 1);`

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

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

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

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

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

1. לבצע import להוק הנכון, במקרה הזה setState.
2. לומר לקומפוננטה בהתחלה איזה משתנה נכנס לזיכרון (במקרה שלנו time) ואת הסוג שלו. להגדיר פונקציה שתשנה את המשתנה בזיכרון. שם הפונקציה הוא תמיד set ואז שם המשתנה מתחיל באות גדולה.
3. לשנות את הקומפוננטה שצריך באמצעות הפונקציה שמשנה את המשתנה.

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

זה כל מה שצריך לעשות.

חשוב לזכור כי לא כדאי להשתמש בסטייט בלי צורך. הסטייט וניהולו הם חלק חשוב מאוד בריאקט ואנו נלמד בהמשך על דרכים שונות לנהל סטייט גלובלי. נוסף על כך, נלמד גם על עוד הוקים.

תרגיל:

צרו קומפוננטה בשם CountDown.tsx שיש בה כפתור ומספר שמתחיל ב-10. הקומפוננטה תספור מהמספר הזה לאחור (אין צורך לעצור ב-0).

פתרון:

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

הקומפוננטה הזו עושה בדיוק אותו הדבר כמו הקומפוננטה של CountUp שיצרנו בפרק, אבל במקום להעלות את count ב-1 היא מורידה אותו ב-1. גם פה כדאי לשים לב לשלושת החלקים:

1. ה-import השונה (מייבאים את ההוק של useState).
2. קובעים את המשתנה שייכנס לזיכרון של הסטייט ואת שם הפונקציה שמכניסה את המשתנה לזיכרון. שם הפונקציה תמיד מורכב מ-set ומשם המשתנה. ניתן לקבוע את הערך ההתחלתי של הסטייט בשלב הזה. לא נשכח להגדיר את סוג המידע של הסטייט.
3. כשרוצים לשנות את המשתנה, משתמשים תמיד בשם הפונקציה שמכניסה את המשתנה לזיכרון.

תרגיל:

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

פתרון:

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

type propsInfo = {
  time:number,
}

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

אני קורא לקומפוננטה הזו כמובן כך:

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

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

שהסטייטים אינם תורה מסיני. הם פשוט משהו ששמים בזיכרון, זה הכול, ואם לא משנים להם את הערך אפשר לגשת אליהם כרגיל.

צריך לזכור כמובן שחשוב להגדיר את סוג המידע כמו שצריך.

olmer@Webnks.com

פרק 10

תכנון מבנה הקומפוננטות

תכנון מבנה הקומפוננטות

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

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

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

type optionsInfo = {
  weekday: string;
  hour: string;
  minute: string;
  second: string;
};

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

זו קומפוננטה אופיינית מאוד לקומפוננטת תצוגה בלבד. אנחנו משתדלים לשמור על הקומפוננטות שלנו קטנות ככל האפשר ולא להכניס לוגיקה ופונקציות מיותרות לקומפוננטת תצוגה. פה היא מקבלת מספר וממירה אותו לתצוגה בעברית. זה הכול.

אם ארצה להשתמש בה, אני אצור קומפוננטה אחרת שמעבירה אליה את הזמן שאני רוצה שקומפוננטת הבת תציג. למשל, קומפוננטת TodayTime.tsx שמעבירה את הזמן של היום אל קומפוננטת Watch.tsx:

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

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

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

נסו את זה בעצמכם. זה קריטי לתרגולים הבאים.

למה השעון לא מתעדכן? אם קראתם את הפרק הקודם – וגם תרגלתם – אתם כבר יודעים שהרִנדור של הקומפוננטה רץ פעם אחת בלבד ושעל מנת לעדכן אותו אנחנו צריכים לעשות משהו – כמו למשל setInterval. מה הבעיה? אני יכול לשים את ה-setInterval בתוך הקומפוננטה, אבל אני לא רוצה לשים אותו ב-Watch. מדוע? לערבב לוגיקה עם קומפוננטת תצוגה זה לא רעיון טוב.

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

את זה ניתן לעשות בקלות עם setInterval:

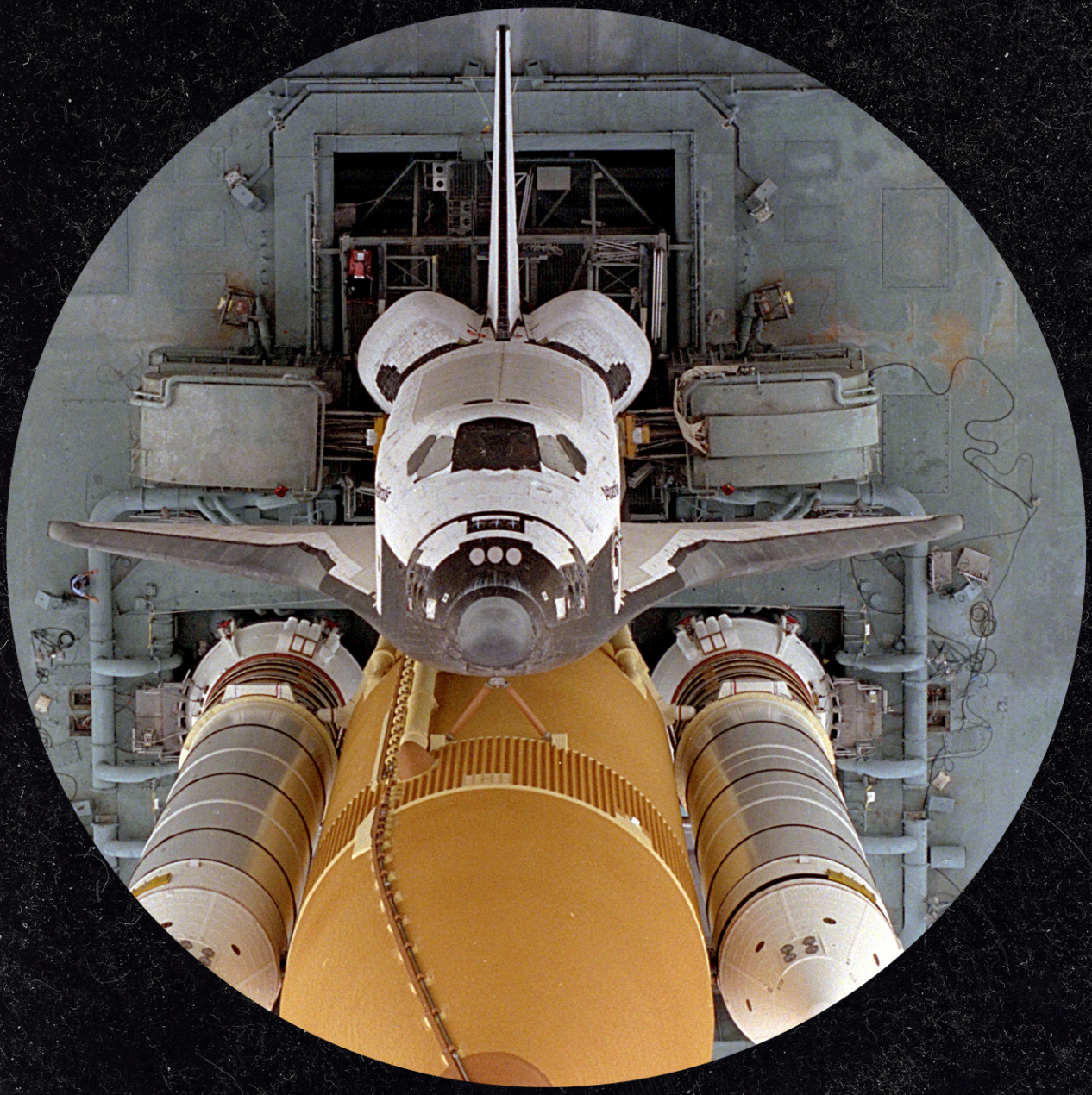
```
import { useState } from 'react';
import Watch from './Watch.tsx';
function TodayTime():JSX.Element {
  const [today, setToday] = useState<number> (Date.now());
  function upTime():void {
    setToday(Date.now());
  }
  setInterval(upTime, 1000);
  return (
    <Watch milliseconds={today} />
  );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

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

otiper@yahoo.com

פרק 11

אירועים ועדכון קומפוננטות



אירועים ועדכון קומפוננטות

עד כה הצגנו קומפוננטות סטטיות או מקסימום כאלו שמתעדכנות כתוצאה מפעילות פנימית כמו setInterval. אבל חלק גדול מהקומפוננטות אמורות לדעת לטפל בקלט מהמשתמש – כלומר פעולות ואירועים. אירועים הם חלק מהותי מג'אווהסקריפט ומג'אווהסקריפט שרץ בסביבת דפדפן, וריאקט יודעת לעבוד איתם יפה מאוד. על מנת לדעת איך עובדים עם אירועים, ניזכר איך אירועי DOM, כלומר אירועים טבעיים בג'אווהסקריפט, עובדים.

אירועי DOM

בג'אווהסקריפט, כאשר אנו עובדים עם אלמנטים טבעיים של HTML, כמו למשל input, אנו יכולים להצמיד להם אירועים. האירועים האלו נקראים אירועי DOM. להלן לוח של כמה אירועים נפוצים ולצידם דוגמאות:

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

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

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

אירועים סינתטיים/ ריאקטיים

ב-JSX אנו יכולים להשתמש באירועים ריאקטיים לאלמנטים שיודעים לעבוד איתם. הם דומים לאירועי ה-DOM וגם מתנהגים כמעט כמוהם. מדובר בעצם באירועים עוטפים שיש להם אותו ממשק 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-camel case, כלומר האות הראשונה של המילה השנייה תהיה אות גדולה באנגלית. למשל:

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

מובן שאני לא מעביר לאירוע מחרוזת טקסט אלא ביטוי בתוך סוגריים מסולסלים על מנת ש-JSX יֵדע להתמודד איתו. מעבר לזה, השימוש באירועים זהה לחלוטין בריאקט. כך, למשל, אם יש לי קומפוננטה שנקראת MyInput ובה אלמנט input של HTML (מדובר באלמנט שמאפשר לנו להקליד טקסט בתוכו), אני יכול להשתמש ב-onChange כדי להציב את מה שהמשתמש כותב בסטייט ומשם לעשות איתו מה שבא לי.

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

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

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

export default InputViewer;
```

איך אנחנו מעבירים את המידע מתוך ה-input אל הסטייט text? בדיוק בשביל זה אנחנו משתמשים ב-onChange, שלפי שמו אנו רואים שהוא אירוע ריאקטי. איך הוא עובד? באמצעות תכונה שמצמידים לאלמנט input. התכונה הזו 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. כלומר הפונקציה הזו:

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

זו פונקציית חץ שפשוט מופעלת עם e. מה זה e? האירוע עצמו, כלומר אובייקט עם מידע על האירוע. יש בג'אווהסקריפט כמה אירועים שאפשר להצמיד לאלמנטים שונים.

הקוד הזה למשל:

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

הוא אירוע onMouseEnter שמתקיים כאשר המשתמש עובר עם העכבר מעל האלמנט המדובר, במקרה הזה input, שדה הטקסט. אם זה קורה, הפונקציה:

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

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

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

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

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

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

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

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

אנו נשתמש באירוע הריאקטי שנקרא onChange ושמופעל בכל פעם שיש שינוי בערך אלמנט ה-input. האירוע יפעיל פונקציה שנקראת changeHandler והיא תבצע אכלוס של הסטייט באמצעות setText שאותו הגדרנו באמצעות ההוק.

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

function InputViewer():JSX.Element {

  const [text, setText] = useState<string>('');
  function changeHandler(e: React.FormEvent<HTMLInputElement>):void
{
    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 שהטקסט מופיע מייד ב-span. מדוע אנחנו מעבירים אל הסטייט רק את e.target.value? כי זה החלק באובייקט האירוע, שכאמור תכונת האירוע מעבירה לנו, שמכיל את המידע שמעניין אותנו. באובייקט האירוע יש מידע רב ואתם מוזמנים לבחון אותו עם הקונסולה או עם הדיבאגר, אבל ב-onChange, המידע הטקסטואלי שיש בשדה מועבר עם e.target.value.

אנחנו יכולים לממש כל התנהגות שאנו רוצים באמצעות אירוע. למשל, נניח שאנו רוצים שהמידע מתוך שדה ה-input יופיע בתוך ה-span רק כאשר אנו מקליקים על כפתור. איך נעשה כזה דבר? ראשית, ניצור כפתור HTML פשוט שיהיה לו אירוע onClick.

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

עכשיו ניצור את הפונקציה שיודעת לעבוד עם האירוע. במקרה הזה מדובר באירוע קליק שלא מכיל מידע, אז איך אני אדע בדיוק מה יש בתוך ה-input? טוב, זה קל – יש לי את אירוע ה-changeHandler שמכניס כל מה שאני מקליד ישירות בשדה ה-input אל משתנה סטייט שנקרא text. אני צריך רק לנתק את ה-span מה-text ולחבר אותו למשתנה סטייט אחר (אקרא לו viewText), והלחיצה תגרום לסטייט viewText להתאכלס מסטייט text.

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

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

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

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

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

export default InputViewer;
```

העניין פה הוא להבין שבסופו של דבר, האירועים בריאקט נראים למשתמש באופן כמעט זהה לאירועי ה-DOM הטבעיים ולא צריך להיבהל מהם. אנו מצמידים את האירועים הריאקטיים לאלמנטי HTML שתומכים בהם כמעט כמו האירועים של ה-DOM, אך בכמה הבדלים קלים - שם הפונקציה שונה (onClick בעטיפה הריאקטית, onclick באירוע הטבעי) וכמובן מה שאנו מעבירים (פונקציית JSX שעטופה ב-{ } בריאקט ומחרוזת טקסט באירוע הטבעי). אם היינו משתמשים בג'אווהסקריפט רגיל, הפעלת האירוע הייתה נראית כך:

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

אבל בגלל ה-JSX, אנו עושים את זה ללא הפעלה ובלי סוגריים מסולסלים.
יש עוד כמה שינויים קלים בין אירוע טבעי לאירוע ריאקטי. באירוע קליק ריאקטי, למשל, אנו בולמים את הפעפוע לא באמצעות return false אלא באמצעות שימוש בפונקציה stopPropagation, אך לא נדון בכך בפרק זה.

אבל איך אנו מצמידים אירועים לקומפוננטה ריאקטית? פה העניינים מסתבכים מעט, אבל ממש מעט. קומפוננטה ריאקטית יכולה להיות למשל קומפוננטה שמכילה input. משהו כזה:

```
function Input():JSX.Element {

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

export default Input;
```

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

אם אני רוצה להשתמש בקומפוננטה Input.tsx בדוגמה שהבאתי קודם לכן, זה לכאורה פשוט – אני רק אחליף את האלמנט input בקומפוננטה Input. משהו בסגנון כזה:

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

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

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

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

export default InputViewer;
```

אבל אם לא תתעצלו ותנסו את הדוגמה בעצמכם, כלומר תעתיקו את Input.tsx לתוך הפרויקט Create React App, תיצרו את קומפוננטת InputViewer.tsx ותציבו אותה באפליקציית הריאקט שלכם ב-app, תראו שזה לא עובד. למה?

כי קומפוננטות ריאקט לא מקבלות אירועים באופן טבעי.

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

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

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

מה בעצם מתרחש כאן?

בקומפוננטה שמשתמשת ב-Input אני לוקח את ה-props.onChange ומעביר אותם לאלמנט ה-input הטבעי. כך, אם אני אשתמש בקומפוננטה Input באופן הבא:

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

היא תעבוד לי.

כיוון שאנו עובדים בטייפסקריפט, אני מגדיר את סוג המידע ש-onChange מחזירה כ-void.

בואו נדגים שוב עם הכפתור באמצעות אחת הדוגמאות הקודמות. אני אצור קומפוננטת כפתור ריאקטית שעוטפת כפתור 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.

קומפוננטת Button.tsx תיראה כך:

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

function Button(props: propsInfo):JSX.Element {

  const clickHandler = props.onClick;

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

export default Button;
```

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

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

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

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

export default InputViewer;
```

זה עלול להיות קצת מבלבל, במיוחד בגלל סוגי המידע המפחידים, אבל כדאי לזכור שמדובר בסופו של יום ב-props שכבר למדנו לעבוד איתם יפה מאוד ושאותם 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 clickHandelr = props.onClick;
6
7     return (
8       <button onClick={clickHandelr}>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 שאנו רוצים ב-props ולהעביר אותו אל האלמנט הטבעי.

תרגיל:

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

רמז: יש צורך להצמיד שני מנהלי אירועים: הראשון הוא onMouseOver, שמכניס לסטייט משתנה כלשהו ואת המילה active, והשני הוא onMouseOut, שמאפס את המשתנה.

פתרון:

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

function MyDivContainer():JSX.Element {

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

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

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

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

export default MyDivContainer;
```

על אותו אלמנט יש לנו (שימו לב שאלמנט div הוא אלמנט HTML טבעי שיכול לקבל אירועים) שני אירועים שונים. האחד, onMouseOver, נכנס לפעולה כאשר העכבר נמצא מעל האלמנט, והשני, onMouseOut, נכנס לפעולה כאשר העכבר יוצא ממנו. כל מה שנותר לי לעשות הוא ליצור את הפונקציות שפועלות בזמן שהאירוע מתרחש. הראשונה לוקחת סטייט שיצרתי ומכניסה לתוכו את המילה active והשנייה מאפסת אותו. אני מכניס את המשתנה שיש בסטייט ועובד איתו.

שימו לב: אפקט מעבר עושים בדרך כלל עם CSS פשוט. כאן אנו עושים את זה עם ריאקט לשם התרגול. נוסף על כך, mouseOver הוא גם אירוע בעייתי שיכול ליצור זליגות זיכרון.

תרגיל:

ארכיטקט המערכת הביט בתרגיל הקודם וביקש שה-div יהיה כקומפוננטת ריאקט. כלומר שהקומפוננטה הקודמת תיראה כך:

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

function MyDivContainer():JSX.Element {

  const [activeText, setActiveText] = useState<string>('');
  function mouseoverHandler():void {
    setActiveText('active');
  }

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

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

export default MyDivContainer;
```

צרו את קומפוננטת Div ב-Div.tsx כדי שהקוד שלעיל יעבוד.

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

פתרון:

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

אפשר גם לקצר את התהליך ולכתוב משהו כזה:

```
import { useState } from 'react';
type propsInfo = {
  onMouseOver: () => void;
  onMouseOut: () => void;
}

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

export default Div;
```

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

MyDivContainer.jsx

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

Div.jsx

src > Div.jsx > ...

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

בעצם בדרך הזו הפונקציות המטפלות באירועים עוברות מאלמנט האב אל ה-props בקומפוננטה שלי, וברגע שהן ב-props אני יכול לשים אותן באלמנט שיש בקומפוננטה הפשוטה שלי.

תרגיל:

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

פתרון:

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

function Counter():JSX.Element {

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

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

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

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

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

export default Counter;
```

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

פרק 12

אלמנט FRAGMENT

אלמנט fragment

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

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

תקבלו הודעת שגיאה:

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

זה לא נראה כמו בעיה גדולה, נכון? כיוון שתמיד אפשר לשים Div או span כאבא. אבל זה כן בעייתי כי במקרים רבים אני לא רוצה שיהיה לי אלמנט אב. למשל, אם אני יוצר קומפוננטות שמחליפות tr, שזה אלמנט HTML שעוטף את השורה בטבלה. נניח משהו כזה:

```
function TableRow():JSX.Element {

  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. המון פעמים אני גם לא רוצה להכניס div או span כי אני לא יודע בתוך איזו קומפוננטה תוצב הקומפוננטה שלי וזה עלול להיות בעייתי. כמה טוב היה אם היינו יכולים להכניס אלמנט אב "שקוף"! אז כן, יש אלמנט כזה שנקרא React.Fragment. האלמנט הזה יכול להיות במקום אלמנט אב והוא לא יודפס כלל. החל מהדוגמאות הבאות אני אשתמש בו.

```
function TableRow():JSX.Element {

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

export default TableRow;
```

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

מקובל גם להשתמש בקיצור <> ו </>.

```
function TableRow():JSX.Element {

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

export default TableRow;
```

פרק 13

USEEFFECTS

useEffects

כפי שלמדנו בפרק על הסטייט, כשקומפוננטה מסוימת משתנה, למשל בגלל אירוע מסוים, היא נבנית מחדש על התצוגה שלה. הבנייה הזו מחדש נקראת רֶנדור. הבה נדגים באמצעות קומפוננטה שנבנתה באחד התרגילים בפרק על אירועים – הקומפוננטה Counter.tsx:

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

function Counter():JSX.Element {

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

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

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

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

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

בפעם הראשונה, ובכל פעם שנעשה שינוי שמשפיע על התצוגה של הקומפוננטה הזו (במקרה הזה עם כפתורים, אבל זה יכול להיות שינוי שנוצר כתוצאה מ-props שהשתנו), מתבצע רִנדור מחדש. לא מעט פעמים אנו רוצים לדעת שהיה רִנדור או רִנדור מחודש של הקומפוננטה. בדיוק בשביל זה אנו יכולים ליצור הוק מיוחד שיפעיל פונקציה שאנו מעבירים לו.

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

הדרך הטובה ביותר להבין את זה היא באמצעות דוגמה:

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

ראשית הבאתי את useEffect כהוק ב-import. הכי קל ונחמד בעולם. השלב הבא הוא פשוט להשתמש בהוק הזה. אני קורא ל- useEffect ומעביר בו ארגומנט שהוא הפונקציה שתפעל. הפונקציה המופעלת לא מחזירה כלום אז אני מגדיר את הפלט שלה כ-void כמו שצריך בטייפסקריפט.

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

מה שירוץ הוא:

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

בכל פעם שהקומפוננטה תרוץ. נסו את זה! השתמשו בקומפוננטה הזו והציצו בקונסולה של כלי המפתחים כאשר אתם משנים 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, משהו בסגנון כזה:

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

useEffect(logEffect);
```

פונקציות חץ עלולות לבלבל, אבל צריך לזכור שהן בסופו של דבר פונקציות רגילות לכל דבר עם כמה תוספות חביבות, כמו למשל שמירה על this. זו הסיבה שכדאי מאוד להשתמש בהן.

אנו משתמשים ב-useEffect למגוון מטרות: בדרך כלל על מנת לקרוא ל-API חיצוני, ללוגים ולפעולות נוספות. תלוי במערכת. useEffect החליפה את "מעגל החיים הריאקטי" שהיה מקובל כשהשתמשו בקומפוננטות מבוססות קלאס. נרחיב על כך בפרק על קומפוננטות מבוססות קלאס. useEffect מכיל פרמטר נוסף שמומלץ להשתמש בו. הפרמטר הזה רלוונטי כאשר אנו רוצים למדוד רִנדור מחודש בעקבות שינוי props (ולא שינוי סטייטים). זהו מערך של התכונות שאחריהן יש לעקוב ורק אם הן משתנות, הפונקציה ב-useEffect תרוץ.

כך, למשל, אם יש לי קומפוננטה מסוימת ואני רוצה להקשיב לרִנדור מחדש שלה שמתבצע רק בגלל שינויי props מסוימים, אני אוסיף כארגומנט שני את ה-props הרלוונטיים:

```
import { useEffect } from 'react';
type propsInfo = {
  count: number;
}

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

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

olim@zivbook.com

פרק 14

קומפוננטה ללא שם

קומפוננטה ללא שם

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

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

function Button(props:propsInfo):JSX.Element {

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

export default Button;
```

קראנו לפונקציה Button ואז ייצאנו אותה. זה סינטקס ולידי שנוח להבנה לאנשים שלומדים ריאקט לראשונה. אבל יש דרך נוספת לכתוב פונקציות. בסופו של דבר, למי שמייבא את הפונקציות ומשתמש בהן אין שום צורך בשם הפנימי הזה והוא רק לצורך הנוחות שלנו. אבל גם אנחנו לא זקוקים לשם הזה. ה-export מקבל גם פונקציות חץ אנונימיות. כלומר כל קומפוננטה שכתבנו עד כה תעבוד באופן מושלם גם אם נמיר אותה לפונקציית חץ אנונימית ונייצא אותה. למשל, הפונקציה Button בהחלט תעבוד כרגיל אם היא תיראה כך:

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

export default (props: propsInfo):JSX.Element => {

  const clickHandelr = props.onClick;

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

זה נראה מפחיד, מבעית ואפילו מוזר לרוב המתכנתים שלא מורגלים בריאקט. אבל אם הגעתם לפרק הזה וקראתם היטב את כל הפרקים - וגם תרגלתם - תוכלו להבין את הקוד הזה אם תיתקלו בו. מה שנעשה כאן הוא פשוט: במקום להצהיר על שם פונקציה, אני יוצר אותה כפונקציה אנונימית ופשוט מחזיר אותה עם export default. הבעיה המרכזית היא שכלי המפתחים של ריאקט לא יציג את שם הקומפוננטה ויקשה עלינו למצוא אותה.

olfer@yahoo.com

פרק 15

עיצוב קומפוננטות

עיצוב קומפוננטות

יש כמה דרכים לעצב קומפוננטות בריאקט, חלקן גם בעזרת ספריות עזר. בפרק זה אנו נתמקד בשתי דרכים עיקריות שבאות יחד עם ריאקט. ריאקט עובדת עם CSS, שהוא סינטקס המאפשר לנו לעצב אלמנטים של HTML שמשתמשים בו בדפי אינטרנט עם או בלי קשר לריאקט. הספר הזה לא מלמד CSS מלבד מבוא קצר, שאמור להספיק לעיצוב בסיסי.

בדפי אינטרנט בסיסיים יש לנו קובצי CSS שאנו מייבאים לדף ה-HTML, אבל באפליקציית ריאקט אנו יכולים להשתמש ב-import, וספריית וובפאק תדאג לטעון את הקובץ יחד עם הקומפוננטה.

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

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

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

כדי להבין איך CSS עובד ואיך הוא עובד בריאקט, אנו נעבוד עם הקומפוננטה Counter.tsx, שבנינו בפרק על אירועים.

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

function Counter():JSX.Element {

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

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

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

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

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

export default Counter;
```

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

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

את הקומפוננטה נציב, כרגיל, ב-App, הקובץ הראשי של אפליקציית הריאקט שלנו. מהרגע הזה, כל שינוי שנעשה ב-CSS יוצג בקומפוננטה. על מנת לבדוק שהכול עובד, נציב בקובץ Counter.css את הטקסט הזה:

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

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

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

CSS בסיסי

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

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

button הוא הסלקטור, background-color הוא התכונה ו-red הוא הערך. אני מעניק את התכונות לכל האלמנטים שמתאימים לסלקטור. במקרה הזה אני מעניק את תכונת צבע הרקע האדום לכל ה-buttons בלי יוצאים מהכלל.

סלקטור

סלקטור יכול להיות מכמה סוגים. הוא יכול להיות אלמנט שלם, 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 הזה לכמה וכמה אלמנטים בקומפוננטה. אם אני רוצה לתת את התכונות האלו ל-myDiv, אני אתן אותן באמצעות הסימן נקודה - ".". כלומר, 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 myDiv וגם לכל ה-buttons, אני אכתוב אותם עם פסיק - "," מפריד ביניהם.

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

סדר הסלקטורים לא משנה בדוגמה הזו, אך הוא משמעותי כאשר יש הוראות סותרות של סלקטורים דומים והדפדפן צריך לקבוע קדימות.

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

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

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

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

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

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

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

אז הסלקטור יהיה:

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

מפני 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 משמעותי מאוד מ-CSS.

תכונות

הסלקטורים קובעים על מי נפעיל את התכונות. התכונות קובעות את העיצוב של האלמנטים: צבע, ממדים ואפילו התנהגות. התכונות נמצאות בתוך הסוגריים המסולסלים שלאחר הסלקטור. הן מורכבות מהתכונה ומהערך שלה. למשל, שם התכונה הוא צבע רקע – background-color והערך הוא red. לכל תכונה יש ערכים שאפשר לתת לה. למשל, התכונה width תקבל ערכים בפיקסלים או ביחידות אחרות.

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

אפשר לתת כמה תכונות שרוצים, ללא הגבלה, כל עוד מקפידים שיהיה סימן נקודה-פסיק (;) לאחר כל תכונה.

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

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

תיאור התכונה	שם התכונה	הערכים שלה
גובה	height	מספר + px, למשל: 100px
רוחב	width	מספר + px, למשל: 100px
צבע טקסט	color	שם הצבע, למשל red, או ערך הצבע בהקס' למשל: #ffffff
צבע רקע	background-color	שם הצבע, למשל red, או ערך הצבע בהקס' למשל: #ffffff
שוליים	margin	מספר + px ארבע פעמים. הערך הראשון קובע את השוליים העליונים, השני את השוליים מימין, השלישי את התחתונים והאחרון את השוליים משמאל. למשל: 10px 10px 10px 10px.
כיוון טקסט בתוך האלמנט	text-align	left או right

עכשיו, כשיש 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 וכמה דרכים נוספות לעיצוב קומפוננטה בריאקט, אבל כל הדרכים האלו נשענות על ספריות צד שלישי. לריאקט עצמה אין דעה בנוגע לדרכי העיצוב, ובדוקומנטציה הרשמית שלה אפשר למצוא את הדרך שפורטה פה. יש דרך נוספת לעיצוב קומפוננטות בשם inline CSS, אך היא אינה מומלצת על ידי הדוקומנטציה ופוגעת בביצועים.

olfer@yahoo.com

פרק 16

קלאס ריאקטי

קלאס ריאקטי

עד כה, כל הקומפוננטות שעבדנו עליהן היו פונקציות. הקומפוננטות האלו נקראות באנגלית functional component והן משתמשות בהוקים על מנת לנהל את הסטייט וגם לדעת מתי הקומפוננטה רונדרה. אבל יש גם קומפוננטות אחרות, קומפוננטות מבוססות קלאס. יש כרגע הנחיה חד משמעית בריאקט שלא לעבוד עם קלאסים והאפשרות הזו תפסיק להתמך בעתיד. כדאי לקרוא את הפרק הזה רק אם אתם עובדים בחברה שבה משתמשים בקוד ישן שיש בו קלאס. במידה ולא, אנא עיברו לפרק הבא.

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 הפונקציה. אבל חשוב לדעת שיש קומפוננטות מבוססות קלאס בריאקט ולו רק בגלל היכולת שלכם לעבוד עם קוד ישן יותר (לגאסי קוד, Legacy Code).

אז איך זה נראה? פשוט מאוד – עם קלאס. קלאס (Class) הוא חלק מג'אווהסקריפט עצמה ומדובר בסינטקס שונה מפונקציה. לקלאס יש כמה חלקים – ראשית, יש לו קונסטרקטור (constructor), הפונקציה הבנאית, שפועלת מייד בהתחלה ושאליה אני גם מקבל את ה-props. הקוד בקונסטרקטור מתבצע ברִנדור הראשוני בלבד. שנית, יש לו render, שיש בו מה שהקומפוננטה מחזירה.
על מנת שהקלאס יהיה קומפוננטה ריאקטית, אנו עושים לו extend לקלאס של React.Component.

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

```
function Welcome():JSX.Element {

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

export default Welcome;
```

המקבילה שלה בקלאס תיראה כך:

```
class Welcome extends React.Component {

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

}

export default Welcome;
```

אפשר לראות כאן שמתודת ה-render, שיש לנו אותה כיוון שהקלאס שלנו יורש מ-React.Component, אחראית על החזרת ה-JSX. עד כאן אין הבדל משמעותי בין קומפוננטת קלאס לקומפוננטת פונקציה, שאותה אנו מכירים.

העניינים נותרים פשוטים גם אם אנו מתחילים להשתמש ב-props. ה-props בקלאס נמצאים על גבי הסקופ של הקומפונטטה, כלומר ב-this.props. בכל מקום נתון בקומפוננטת הקלאס אני יכול להשתמש ב-this.props ולקבל גישה ל-props.
כדי להדגים, הבה נניח שאנו רוצים להוסיף את שם המשתמש ולקבל אותו מה-props. בקומפוננטת פונקציה נעשה משהו כזה:

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

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

בקלאס אנו ניגשים אל ה-props באמצעות ה-this. אבל למעט העובדה הזו, אין שינוי מהותי. על מנת לממש את אותה התנהגות בדיוק עם קלאס, אני אכתוב את הקוד הבא:

```
class Welcome extends React.Component {

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

}

export default Welcome;
```

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

הבה נניח שאנו רוצים ליצור סטייט עם ברכה – כלומר לקבל את ה-props.name, ליצור ברכה ולהכניס אותה לסטייט. אם אני ארצה לעשות את זה עם פונקציה, רק לשם הדוגמה, אני אעשה משהו כזה:

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

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

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

export default Welcome;
```

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

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

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

}

export default Welcome;
```

אפשר לראות שהשתמשתי בקונסטרקטור. הקונסטרקטור מקבל props וגם מפעיל את super. השורה הזו חשובה ואפילו קריטית כיוון שהיא קוראת לקונסטרקטור של React.component. זו לא המצאה של ריאקט אלא חלק מהסינטקס של ג'אווהסקריפט. בלי השורה הזו, הקונסטרקטור שלכם לא יעבוד כשורה.

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

אין לנו הוקים ואנו לא מגדירים פונקציית שינוי סטייט. הפונקציה שאנו משתמשים בה לשינוי הסטייט היא setState שמקבלת אובייקט סטייט. השימוש שלה ייראה כך:

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

זה נראה מורכב יותר מאשר המימוש הנאה והנקי יותר של הפונקציה, וזה נכון. מכיוון שיש לנו קלאס, אנחנו נכנסים גם לבעייתיות של קונטקסט.

הבה נדגים על ידי המרת הקומפוננטה הזו, שהיא פונקציה המשתמשת באירועים, לקלאס:

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

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

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

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

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

export default Counter;
```

זו קומפוננטת פונקציה עם אירועים שעשינו באחד התרגילים בפרק על קומפוננטות ואירועים. אם נרצה להמיר אותה לקלאס, ראשית נזכור ליצור קונסטרקטור ולשנות את הסטייטים, אבל יש לנו גם אירועים, ולאירועים אנו חייבים להעביר קונטקסט של this. אחרת, בשעת הפעלת האירוע תעבוד הפונקציה בסקופ אחר (הסקופ של החלון) ולא תהיה לה גישה לסטייט.

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

export default Counter;
```

צריך לשים לב היטב לשורות האלו בקונסטרקטור:

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

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

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

הסקופ הוא לא של הקלאס. על מנת לגרום לאירוע לעבוד בתוך הקלאס עם נגישות למשתנים, למתודות וכמובן לסטייט, יש צורך להשתמש ב-bind. ה-bind גורם לאירוע לעבוד בסקופ של הקלאס. אם ה-bind לא יהיה קיים, אני אקבל שגיאה כזו:

```
TypeError: Cannot read property 'setState' of undefined
```

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

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

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

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

מעגל החיים בקלאס

בקלאס ניתן גם לשלוט במעגל החיים של הקומפונטטה. בעוד בפונקציה יש לנו רק את ההוק useEffect, שפועל כאשר הקומפוננטה מתרנדרת בפעם הראשונה וכאשר היא מתרנדרת מחדש, בקלאס יש לנו שליטה מלאה יותר במעגל החיים וקיימות לא פחות משש מתודות לעשות זאת.

componentDidMount

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

componentDidUpdate (prevProps, prevState, snapshot)

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

שימו לב: בניגוד לcomponentDidMount, שנקרא רק פעם אחת במעגל החיים של קומפוננטה, componentDidUpdate נקרא לאחר כל פעם שמתבצע רִנדור (חוץ מהפעם הראשונה, שבה componentDidMount מטפל). כדאי גם לשים לב שמפני שהמתודה הזו 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 לניקיונות של interval או ללוגינג.

shouldComponentUpdate (nextProps, nextState)

מתודה שבה אנו יכולים לקבוע אם אנו בכלל רוצים שהקומפוננטה תתרנדר מחדש. היא מופעלת כאשר ה-props או הסטייט מתעדכנים. המתודה הזו אמורה להחזיר true אם אנו רוצים רִנדור מחודש או false אם אנו לא רוצים רִנדור כזה. היא נדירה יחסית ומאפשרת לנו גישה אל ה-props שהתעדכנו ואל הסטייט החדש. היא מתרחשת לפני componentDidUpdate וכאמור, אם היא מחזירה false, אז componentDidUpdate לא תרוץ כיוון שהקומפוננטה לא תרונדר. התכנון

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

היא נדירה יחסית כי ברובם המוחלט של המקרים אנו נסמוך על המנוע של ריאקט שיקבע אם קומפוננטה תרונדר או לא. אנו נשתמש בה כאשר נרצה לשפר ביצועים כיוון שאם חוסכים ברִנדור, מקבלים ביצועים משופרים לכל הקומפוננטות.

getDerivedStateFromError (error)

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

componentDidCatch (error, info)

בדומה ל-getDerivedStateFromError, גם המתודה הזו מופעלת כאשר אחת מתת-הקומפוננטות זורקת שגיאה, אך היא יכולה גם לקרוא לפונקציות אחרות (כמו לוגינג). חלקן מיועדות לשימושי קצה, אבל חלקן שימושיות מאוד בקומפוננטות מסוימות. כך, למשל, בקומפוננטות שיש בהן interval, מומלץ מאוד להשתמש דווקא בקומפוננטות קלאס כי אפשר לנקות את ה-interval אחר כך ולהקל על הזיכרון. אם לא נעשה את זה, ה-interval ימשיך לרוץ גם אם הקומפוננטה נמחקה או אם עברנו למקום אחר, וזה יכול לגרור זליגת זיכרון ובעיות ביצועים.

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

```
import React from "react";

interface TodayTimePropsInfo {
  interval: Function;
}

interface TodayTimeState {
  time: number;
}

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

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

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

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

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

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

```
export default TodayTime;
```

הפונקציה componentWillUnmount בעצם מופעלת אוטומטית כאשר הקומפוננטה נמחקת מה-DOM, וזה קורה כתוצאה מפעולות שונות, כמו מעבר דף בראוטינג, שעליו נרחיב בשלב מאוחר יותר.

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

```
import React from "react";

interface ErrorBoundaryPropsInfo {
  children: React.ReactNode;
}

interface ErrorBoundaryState {
  hasError: boolean;
}

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

עיון בקומפוננטה מראה שהיא תופסת כל שגיאה ומציגה אותה. אם לא – היא מציגה את מה שיש בתוכה. זה מקובל מאוד ואת זה, נכון לגרסה האחרונה של ריאקט, אפשר לעשות רק בקלאס.

תרגיל:

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

פתרון:

```
import React from "react";

interface TodayTimePropsInfo {
  interval: Function;
}

interface TodayTimeState {
  time: number;
}

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

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

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

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

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

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

export default TodayTime;
```

אין כאן משהו שאינכם מכירים. את הסטייט אנו מגדירים כבר בקונסטרקטור ואנו מאתחלים אותו כ-0, ויצרנו handler לקליק שמעלה את הסטייט ב-1.

כיוון שהאירוע פועל בסקופ אחר, אנו חייבים לקשור אליו את הסקופ שלנו באמצעות:

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

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

תרגיל:

מנהל המוצר של בזק פנה אליכם בבקשה ליצור קומפוננטה שמציגה התקפות סייבר. המספר שלהן מתחיל ב-1,000 ועולה ב-10 בכל שנייה. עליכם לממש זאת עם קלאס כיוון שאתם צריכים להשתמש ב-setInterval ויש צורך לנקות אותו עם clearInterval.

פתרון:

```
import React from "react";

interface CurrentCyberAttackCounterPropsInfo {
  // props of CurrentCyberAttackCounterPropsInfo
}

interface CurrentCyberAttackCounterState {
  attackNumber: number;
}

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

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

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

אנו יודעים שאנו צריכים להשתמש בקלאס אם יש צורך להשתמש בניקיון אחרי שהקומפוננטה נעלמת.
ראשית, אנו משתמשים בקלאס כרגיל. כיוון שיש לי צורך בזיכרון פנימי, אני יודע שאני צריך סטייט ואני מגדיר סטייט כבר בקונסטרקטור באמצעות this.state.

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

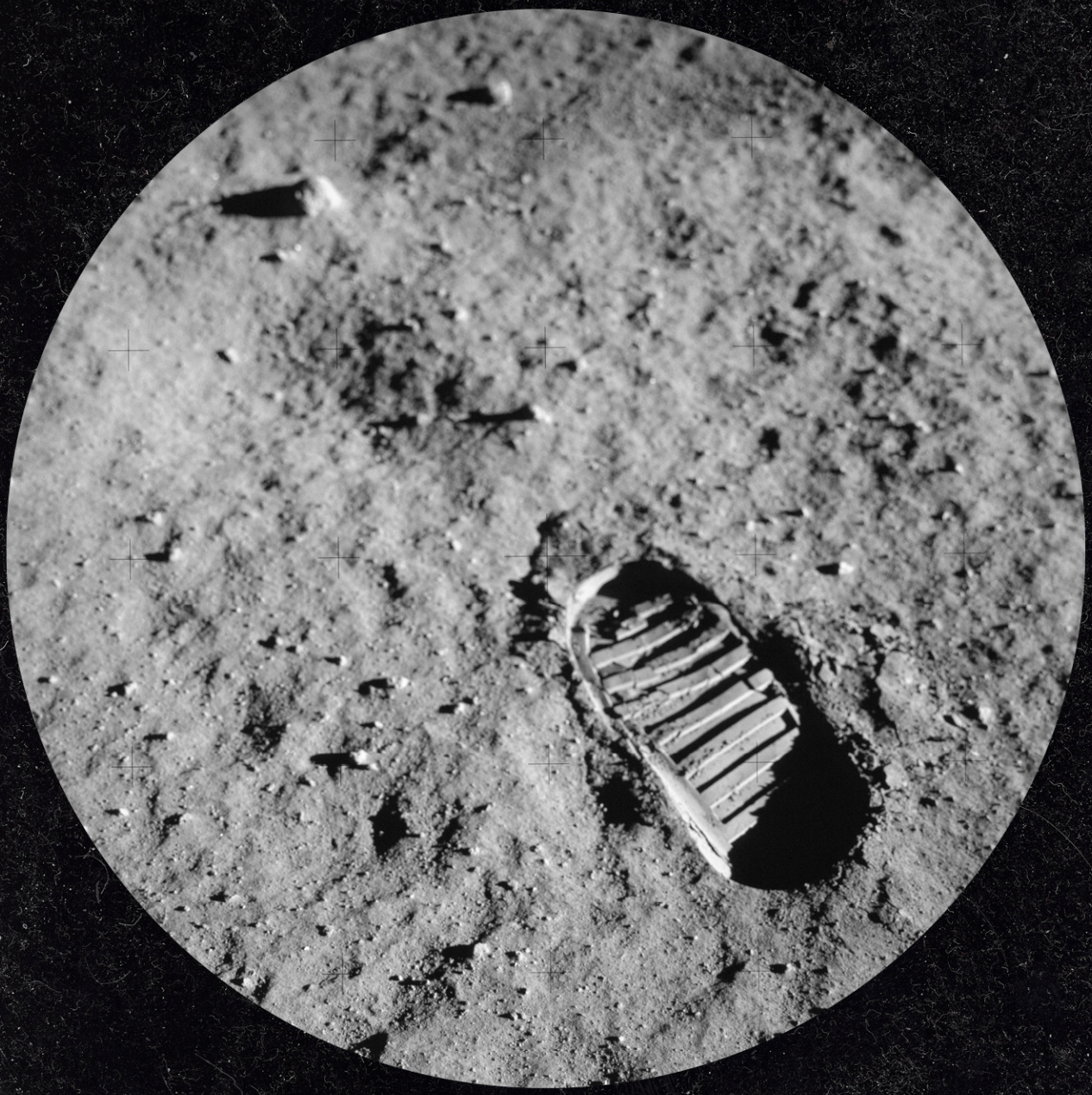
```
this.timerTick = this.timerTick.bind(this);
```

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

otherworld.co.il

פרק 17

שימוש בקומפוננטות ממקורות אחרים



שימוש בקומפוננטות ממקורות אחרים

אחת מהחוזקות הגדולות של ריאקט היא שימוש בספריות אחרות של קומפוננטות. כך בעצם אנחנו לא נדרשים לבנות מאפס 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 ואוספי קומפוננטות שקל לקחת וליצור איתם אפליקציות מורכבות ביותר ויפות מאוד מבחינה ויזואלית.

הספריות האלו באות כחבילות תוכנה שאנו יכולים להתקין בפרויקט שלנו, שיצרנו ב- Create React App. הפרויקט הזה מכיל לא מעט חבילות תוכנה, כמו babel למשל, והחבילות האלו מנוהלות על ידי מנהל התוכנה של Node.js (ג'אווהסקריפט בסביבת השרת) שנקרא npm. לא נדרש לדעת Node.js כדי להשתמש ב-npm ולהתקין חבילות תוכנה (או להסיר אותן).
על מנת להתקין חבילת 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 ונלחץ על New Terminal. ייפתח לנו ממשק של שורת פקודה בתחתית העורך ונוכל לבצע התקנה באמצעות npm.

לחלופין, אם אין לכם, מסיבה מסוימת, Visual Studio Code, אתם יכולים, אם יש לכם מחשב מבוסס חלונות, להגיע למצב cmd ואז לנווט באמצעות הפקודה cd אל תיקיית הפרויקט שלכם. גם שם אפשר לבצע התקנה כזו.

כדי להתקין קומפוננטות ממקומות אחרים אנחנו צריכים לדעת אילו קומפוננטות אנו רוצים. יש אלפי אוספי קומפוננטות וקומפוננטות שונות. מקור טוב לחפש הוא גוגל. כאן נתרגל באמצעות React Material UI – אוסף קומפוננטות בעיצוב חברת גוגל שיצר המון קומפוננטות בסיסיות בריאקט. העיצוב הזה נפוץ מאוד ולא מעט אפליקציות ווביות ואתרים משתמשים באוסף הזה.
כתובת הפרויקט היא:
https://material-ui.com/

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

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

הפקודה הזו משתמשת ב-npm על מנת להוריד ולהוסיף לפרויקט באופן אוטומטי את ספריית React Material UI 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
```

עכשיו אפשר להשתמש בקומפוננטות האלו.

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

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

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

אחת הקומפוננטות היא קומפוננטת כפתור בשם Button, שבעצם יוצרת כפתורים מעוצבים. הבה נשתמש בה עם הקומפוננטה Counter שבנינו בעבר:

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

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

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

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

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

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

export default Counter;
```

אם תריצו את הקומפוננטה הזו, תוכלו לראות שהכפתורים הם כפתורים רגילים ומכוערים. הבה ניצור כפתורים מעוצבים יותר. איך? יש בספרייה של React Material UI כפתורים יפים הרבה יותר. שם הרכיב של הכפתור הוא Button. אנו מייבאים אותו באמצעות:

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

ומשתמשים בו בדיוק 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 לבלבל או להטריד בתחילה, אבל זה פשוט שם החבילה (אפשר להשתמש בתווים מיוחדים בשם חבילה). זה הכול!

בדוגמה אפשר לראות שהעברתי props בשם variant ו-color שמשפיעים על מראה הכפתורים. בתיעוד של הקומפוננטה של הכפתור, וכל קומפוננטה אחרת, יש הסברים על ה-props שאפשר להעביר ועל איך הם משנים את המראה של הקומפוננטה.

אפשר, כמובן, להשתמש בכמה קומפוננטות שרוצים באותה קומפוננטה, בדיוק כמו בקומפוננטה רגילה!

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

function InputViewer():JSX.Element {
  const [text, setText] = useState<string>('');
  const [viewText, setViewText] = useState<string>('');
  function changeHandler(e: React.FormEvent<HTMLInputElement>):void
{
    setText(e.target.value);
  };

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

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

export default InputViewer;
```

ייבוא כמה קומפוננטות

אם אני מייבא כמה קומפוננטות מאותה ספרייה, מקובל להשתמש בסינטקס מעט שונה לייבוא.
במקום:

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

כותבים:

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

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

חשוב לציין שזה-זה הכוח האמיתי של ריאקט – השימוש בספריות קומפוננטות כאלו, שנותנות לנו כוח אדיר רק באמצעות התקנה פשוטה של ספריות קוד פתוח ושימוש קל. באתר של React Material UI יש דוגמאות והסברים רבים כיצד להשתמש בקומפוננטות, ובזמן קצר מאוד אפשר לבנות דפים מורכבים מאוד.

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

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

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

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

השלב הבא הוא פשוט... להשתמש בה!

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

function TableViewer():JSX.Element {

    return (
        <Grid
            rows={[
                { id: 0, name: 'Avraham', city: 'Aram Naharaim' },
                { id: 1, name: 'Itzhak', city: 'Desert' },
                { id: 2, name: 'Yaakov', city: 'Tent' },
                { id: 3, name: 'Esav', city: 'Field' },
                { id: 4, name: 'Moshe', city: 'Cairo' },
            ]}
            columns={[
                { name: 'id', title: 'ID' },
                { name: 'name', title: 'Name' },
                { name: 'city', title: 'City' },
            ]}>
            <Table />
            <TableHeaderRow />
        </Grid>
    );
}

export default TableViewer;
```

אם תנסו את הקוד הזה, תראו שנוצרת לכם טבלה מעניינת.

יש לא מעט מודולים חיצוניים שיכולים לחסוך לכם המון עבודה ומאפשרים לכם לקבל תוצאות מהירות ויפות כמעט מייד. שווה להתחיל להתעניין ולנסות, למשל, את React Material UI. יש הרבה דמואים ודוגמאות קוד מעניינות שדרכם אתם יכולים ליצור תוצר מהמם במעט מאוד עבודה.

מודולים חסרים

לעיתים, כשתנסו קומפוננטות מספריות אחרות, אתם עלולים להיתקל בשגיאה נוסח:

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

או בשגיאה דומה. השגיאות האלו מתקבלות כאשר המודול פשוט לא הותקן על ידי npm. במקרה הזה פשוט כדאי לגגל את התוצאה או להתקין את הספרייה. בשגיאה הזו הספרייה החסרה היא @material-ui/icons והבעיה נפתרת על ידי הקלדה בטרמינל של השורה:

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

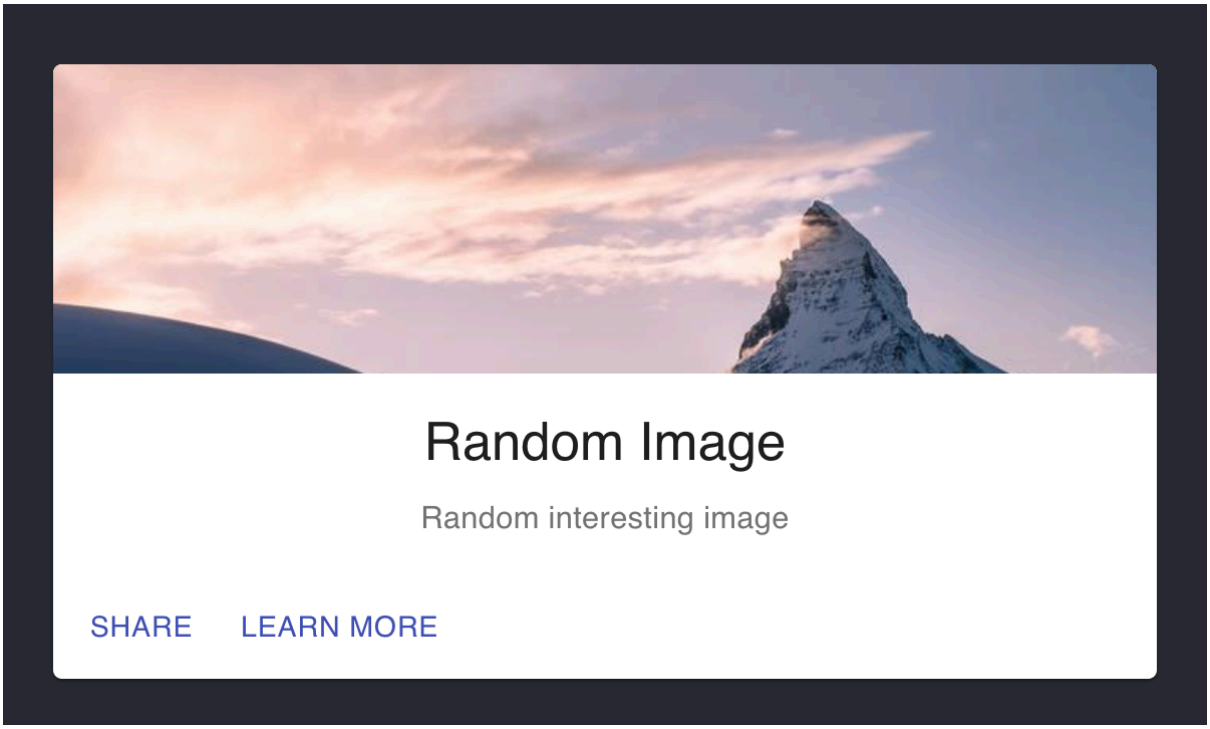
היא מתקינה את הספרייה החסרה שצריך להשתמש בה.

תרגיל:

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

```
https://picsum.photos/id/866/700/400
```

צרו את הרכיב הזה:



רמז: התיעוד והדוגמאות לכרטיס נמצאים בכתובת הזו:

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

פתרון:

```
import Card from '@mui/material/Card';
import CardActionArea from '@mui/material/CardActionArea';
import CardActions from '@mui/material/CardActions';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Button from '@mui/material/Button';
import Typography from '@mui/material/Typography';

export default ():JSX.Element => {
  return (
    <Card sx={{ maxWidth: 345 }}>
      <CardActionArea>
        <CardMedia
          component="img"
          image="https://picsum.photos/id/866/700/400"
          title="Random Image"
          height="140"
        />
        <CardContent>
          <Typography gutterBottom variant="h5" component="h2">
            Random Image
          </Typography>
          <Typography variant="body2" color="textSecondary"
component="p">
            Random interesting image
          </Typography>
        </CardContent>
      </CardActionArea>
      <CardActions>
        <Button size="small" color="primary">
          Share
        </Button>
        <Button size="small" color="primary">
          Learn More
        </Button>
      </CardActions>
    </Card>
  );
}
```

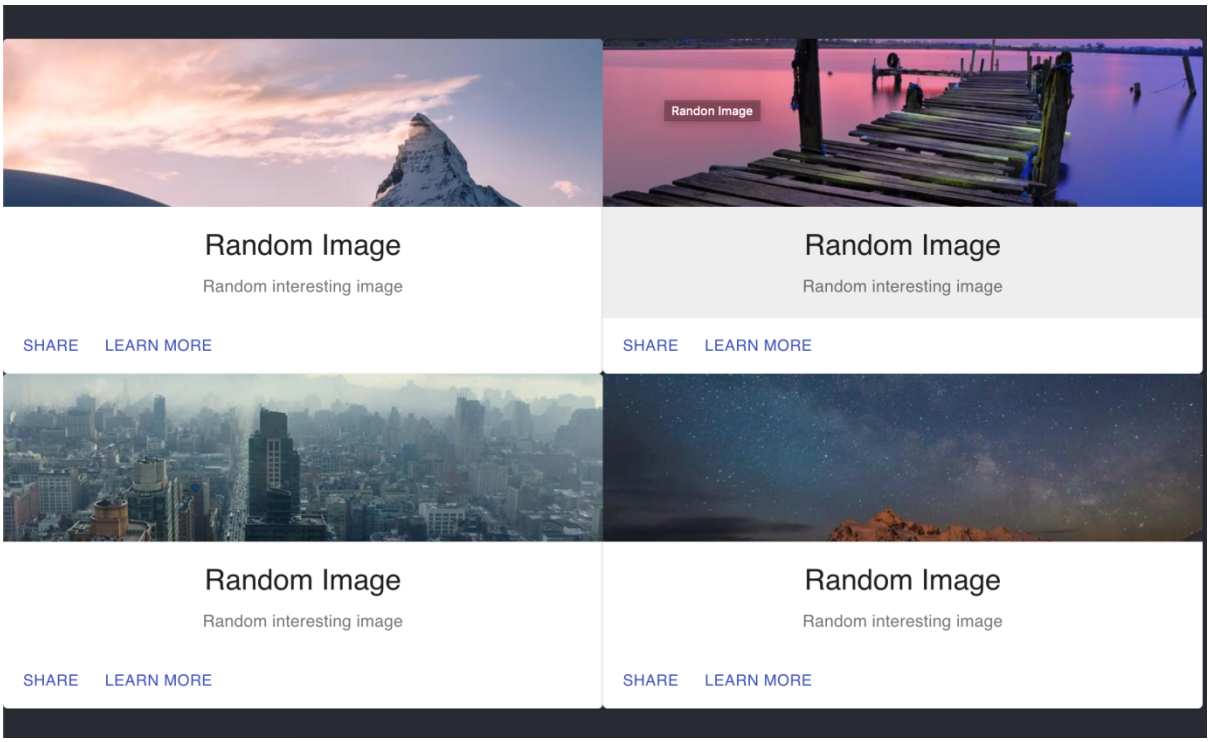
אין כאן גאונות גדולה, פשוט העתקתי את הדוגמה המופיעה בדוקומנטציה כדי ליצור את מה שרציתי ליצור. במקרים האלו אין מה לשבור את הראש ולהתאמץ, אלא צריך פשוט לקחת את מה שיש בדוקומנטציה.

כדאי לשים לב שהדרך של React Material UI לקביעת עיצוב היא דרך שונה המשתמשת ב-useStyles, שזה 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 לשים ב-RandomImageCard.tsx, ואז להשתמש בקומפוננטה הזו איפה שרוצים או לדאוג להעביר לה props שיקבעו את התמונה השונה.

תרגיל:

באמצעות הרכיב הקודם, צרו ארבעה כרטיסים כאלו:



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 בדיוק כפי שלמדנו.

פתרון:

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

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

אמנם הקומפוננטה נראית מורכבת, אבל השינוי הזה אינו מורכב:

```
import Card from '@mui/material/Card';
import CardActionArea from '@mui/material/CardActionArea';
import CardActions from '@mui/material/CardActions';
import CardContent from '@mui/material/CardContent';
import CardMedia from '@mui/material/CardMedia';
import Button from '@mui/material/Button';
import Typography from '@mui/material/Typography';

export default (props :{imageSrc:string}):JSX.Element => {
  return (
    <Card sx={{ maxWidth: 345 }}>
      <CardActionArea>
        <CardMedia
          component="img"
          image={props.imageSrc}
          title="Random Image"
          height="140"
        />
        <CardContent>
          <Typography gutterBottom variant="h5" component="h2">
            Random Image
          </Typography>
          <Typography variant="body2" color="textSecondary"
component="p">
            Random interesting image
          </Typography>
```

```
        </CardContent>
      </CardActionArea>
      <CardActions>
        <Button size="small" color="primary">
          Share
        </Button>
        <Button size="small" color="primary">
          Learn More
        </Button>
      </CardActions>
    </Card>
  );
}
```

השורה שהשתנתה היא:

```
image={props.imageSrc}
```

כעת רק נותר להעביר לקומפוננטה את הפרמטרים האלו ולעשות את זה ארבע פעמים בקומפוננטה מסוימת, למשל ב-**MyContainer**:

```
import RandomImageCard from './RandomImageCard';
import { Grid } from '@mui/material';

function MyContainer():JSX.Element {

    return (
        <Grid container xs={12}>
            <RandomImageCard
imageSrc="https://picsum.photos/id/866/700/400" />
            <RandomImageCard
imageSrc="https://picsum.photos/id/867/700/400" />
            <RandomImageCard
imageSrc="https://picsum.photos/id/868/700/400" />
            <RandomImageCard
imageSrc="https://picsum.photos/id/869/700/400" />
```

```
        </Grid>
    );
}

export default MyContainer;
```

השתמשתי כאן ב-Grid, שגם היא קומפוננטה של React Material UI, כדי לסדר את העיצוב. תוכלו לבדוק אותה בדוקומנטציה. אם השתמשתם ב-CSS זה מעולה, אבל קל יותר להשתמש בקומפוננטה מסודרת שמישהו כבר יצר.

כל שנותר לי הוא לקחת את קומפוננטת MyContainer ולהציב אותה איפה שאני רוצה, למשל ב-app:

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

function App():JSX.Element {

  return (
    <div className="App">
    <header className="App-header">
      <MyContainer />
    </header>
    </div>
  );
}

export default App;
```

ועכשיו אני יכול לבחון בהנאה את התוצאה.

לא צריך ליצור כל קומפוננטה לבד. מומלץ מאוד וגם צריך להשתמש בקומפוננטות של אחרים כדי ליצור ממשקים מהממים. לכל קומפוננטה וקומפוננטה יש API משלה. כך, למשל, ב React Material UI- אפשר להעביר onClick כדי לבצע פעולה מסוימת, כמו לפתוח טאב חדש.

תרגיל:

נוסף על פרמטר התמונה, העבירו גם פרמטר של קישור. לחיצה על Learn More תפתח את הקישור הזה.

רמז: פתיחת קישור בלשונית דפדפן חדשה נעשית באמצעות:

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

פתרון:

```
import Card from "@mui/material/Card";
import CardActionArea from "@mui/material/CardActionArea";
import CardActions from "@mui/material/CardActions";
import CardContent from "@mui/material/CardContent";
import CardMedia from "@mui/material/CardMedia";
import Button from "@mui/material/Button";
import Typography from "@mui/material/Typography";

export default (props: { linkTo: string; imageSrc: string }):
JSX.Element => {
  function goTo(): void {
    window.open(props.linkTo);
  }

  return (
    <Card sx={{ maxWidth: 345 }}>
      <CardActionArea>
        <CardMedia
          component="img"
          image={props.imageSrc}
          title="Random Image"
          height="140"
        />
        <CardContent>
          <Typography gutterBottom variant="h5" component="h2">
            Random Image
          </Typography>
          <Typography variant="body2" color="textSecondary"
component="p">
            Random interesting image
          </Typography>
        </CardContent>
```

```
      </CardActionArea>
      <CardActions>
        <Button size="small" color="primary">
          Share
        </Button>
        <Button onClick={goTo} size="small" color="primary">
          Learn More
        </Button>
      </CardActions>
    </Card>
  );
};
```

אנו צריכים לזכור לא להתבלבל ולא לפחד משפע הקומפוננטות שיש פה. יש כפתור? מצוין, אפשר להעביר אליו onClick, בדיוק כפי שלמדנו. הכפתור, שהוא רכיב של React Material UI, יודע יפה מאוד לטפל באירוע. אנו מעבירים לו אירוע פשוט שפותח מה שאנו מעבירים לו ב-props. במקרה הזה:

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

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

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

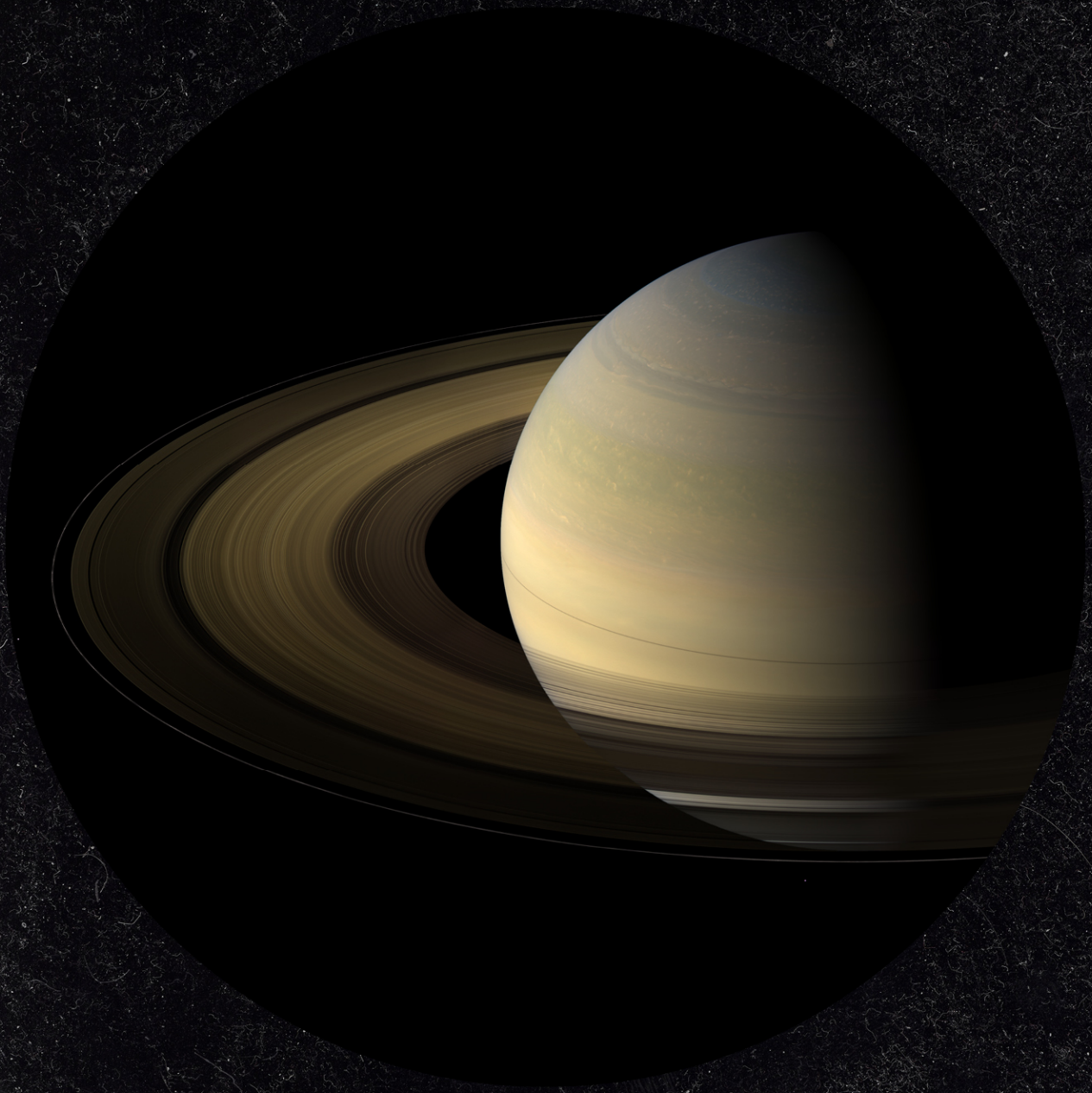
כך בעצם אני יכול להמשיך ולפתח על בסיס הקומפוננטה.

שימו לב שמקובל, במקום לחזור על קוד, להשתמש בפונקציית map.

offer@webit.com

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 המועשרת. ההעשרה הזו מתבצעת באמצעות פונקציה עוטפת שמקבלת כארגומנט קומפוננטה. הפונקציה שמבצעת את השינוי נקראת **HOC**, ראשי תיבות של Higher Order Component. המשמעות של המושג הזה היא "קומפוננטות מדרג גבוה" – כלומר כאלו שלא מלכלכות את הידיים בפונקציונליות אלא מנפקות קומפוננטות אחרות, שהן אלו שעושות את העבודה.

האמת היא שכמתכנתי ריאקט מתחילים לא יצא לכם לכתוב הרבה HOC, אך בוודאי יצא לכם להשתמש בהם והשימוש בהם עלול להיות מבלבל או סתם יכול להיראות כמו וודו – כלומר כישוף אפל שלא ברור איך הוא עובד.

כשאנו כותבים HOC, אנו צריכים לזכור שהקלט הוא קומפוננטה והפלט הוא קומפוננטה אחרת שמשתמשת בקומפוננטה של הקלט כבסיס. משהו בסגנון הזה:

```
const HighOrderComponent = (WrappedComponent:JSX.Element) => {
  function HOC():JSX.Element {
    // Stuff
    return <WrappedComponent />;
  }
  return HOC;
};

export default HighOrderComponent;

const SimpleHOC:JSX.Element = HighOrderComponent(MyComponent);
```

מה קורה בפונקציה הזו? כיצד אני מעשיר את הקומפוננטה? הבה נדגים עם קומפוננטה שאני רוצה להעשיר אך ורק בתכונה אחת שאני רוצה להעביר לה. איזו? משמעות החיים, הקיום וכל

השאר. ערך התכונה הזו, כפי שיודע כל מי שקרא את הספר "מדריך הטרמפיסט לגלקסיה", הוא 42. איך אני אעשיר את הקומפוננטה שלי? אני אעביר אליה את כל ה-props שהיא מקבלת (אני לא יודע אילו) וגם את התשובה לחיים עצמם. בדיוק כך:

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

export default HighOrderComponent;

const SimpleHOC: JSX.Element = HighOrderComponent(MyComponent);
```

הדבר שהכי מפחיד מתכנתים צעירים הוא ה-spread operator:

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

אבל מה שהוא עושה זה לקחת את כל ה-props שמועברים ולהעביר אותם, איך שהם, בפורמט מסודר, אל הקומפוננטה שאני עוטף, בלי שאני אצטרך להכניס אותם ידנית. נוסף על כך, אני מעביר גם את משמעות החיים.

זו דוגמה מאוד תיאורטית, אבל היא מסייעת להבין עד כמה HOC הם פשוטים למרות הסינטקס המאיים. פשוט לוקחים קומפוננטה אחת ומוסיפים לה תכונות או מתודות. הבה נראה דוגמה מעשית יותר. באחד הפרקים הקודמים יצרנו קומפוננטה שמכילה סטייט עם מספר, שאותחל ל-0, ולחיצה על הכפתור גרמה למונה לעלות ב-1. מה הבעיה? בכל פעם שביצענו רענון לדף, הסטייט

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

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

function CounterUp():JSX.Element {

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

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

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

export default CounterUp;
```

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

שמירה ב"זיכרון" של הדפדפן, באופן שישרוד גם רענון, נעשית באמצעות שמירה לאובייקט שנקרא localStorage. מדובר בקובץ קטן שנשמר על המחשב ושומר נתונים גם לאחר רענון של הדפדפן. הכתיבה אליו מתבצעת באמצעות:

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

והקריאה מתבצעת באמצעות:

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

ראוי לציין שאין שום קשר בין localStorage לריאקט. מדובר בפיצ'ר של הדפדפן ושל הג'אווהסקריפט שרץ עליו, בדיוק כמו fetch, למשל, המשמש ל-AJAX בסביבת שרת.

פונקציית ה-HOC שלי תיקח כל קומפוננטה ותוסיף לה מתודות save ו-load. לפונקציה אני אקרא WithStorage:

```
const WithStorage = (WrappedComponent:JSX.Element) => {
    function HOC(props):JSX.Element {
        function save(key:string, data:any) {
            localStorage.setItem(key, data);
        }
        function load(key:string):any {
            return localStorage.getItem(key);
        }
        return <WrappedComponent
            {...props}
            save={save}
            load={load}
        />;
    }
    return HOC;
};

export default WithStorage;
```

מה היא עושה? מוסיפה לכל קומפוננטה שעוברת דרכה שתי מתודות פשוטות: save ו-load. מתודת save שומרת ל-localStorage ומתודת load טוענת מה-localStorage.

ואיך אני מעשיר קומפוננטה כזו? אני פשוט זוכר שהקומפוננטה שעוברת דרך ה-HOC מקבלת ל-props שלה כמה תכונות חדשות, במקרה שלנו save ו-load.

ראשית, הטעינה עצמה נראית כך:

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

const ComposedCompnent:JSX.Element = WithStorage(CounterUp);

function MyContainer():JSX.Element {
    return (
        <ComposedCompnent />
    );
}

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 ComposedCompnent :JSX.Element = WithStorage(CounterUp);
```

בקומפוננטה הזו אני יכול להשתמש כמו בכל קומפוננטה אחרת. צריך לזכור שה-HOC, במקרה הזה WithStorage, תמיד מחזיר לי קומפוננטה שאני יכול להשתמש בה ב-JSX. זה בדיוק מה שאני עושה בקומפוננטת הקונטיינר:

```
function MyContainer() :JSX.Element {
    return (
        <ComposedCompnent />
    );
}
```

אז אחרי שהעשרתי, אני יכול להשתמש ב-save וגם ב-load בקומפוננטה המקורית שלי. איך? פשוט מאוד – ה-HOC מוסיף אותן דרך ה-props, אז אני יכול להשתמש בהן דרך ה-props. הינה למשל ה-save:

```
import { useState, useEffect } from 'react';
import Button from '@mui/material/Button';
function CounterUp(props :{save :(key:string, value:number)=>void})
:JSX.Element {
  const [count, setCount] = useState<number>(0);

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

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

export default CounterUp;
```

בכל פעם שה-state מתעדכן, ב-increasehandler שמופעל לאחר לחיצה על הכפתור, אני קורא ל-props.save ומעביר דרכו שני פרמטרים: ה-key שהחלטתי שהוא ה-counter והערך, שהוא מספר.

שימו לב: על מנת למנוע שגיאה אם הקומפוננטה לא עטופה ב-HOC, הוספתי תנאי שבודק אם יש לנו props.save.

טוב, השמירה היא קלה, אבל מה עם החילוץ? פה אני חייב להשתמש ב-**useEffect**, ההוק שרץ בכל רִנדור. אני רוצה שהוא ירוץ אך ורק ברִנדור הראשון, אז הארגומנט השני שלו הוא מערך ריק. ב-useEffect, שקורה רק כאשר הקומפוננטה מתרנדרת לראשונה, אני אקרא למתודת load עם המפתח (שהוא counter), אמיר אותו למספר (כל הערכים ב-localStorage נשמרים כמחרוזת טקסט) ואכניס אותו לסטייט. אם אין כלום ב-localStorage, אני מחזיר מספר.

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

type propsInfo ={
  load: (value: string) => void;
  save: (key: string, value: number) => void;
}

function CounterUp(props: propsInfo) :JSX.Element {
  useEffect(() => {
    let counter:any;
    if(props.load) {
      counter = props.load('counter');
    }
    const initialCounter:number = Number(counter) || 0;
    setCount(initialCounter);
  }, []);

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

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

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

```
}

export default CounterUp;
```

קטע הקוד של useEffect עלול לבלבל. הבה ננתח אותו שוב. מדובר בפונקציה שמקבלת שני פרמטרים: פונקציה אנונימית ומערך ריק. המערך הריק נועד להורות לריאקט להפעיל את הפונקציה האנונימית רק פעם אחת, בתחילת טעינת הקומפוננטה.

בפונקציה האנונימית יש קריאה ל-props.load, המתודה שניתנה לנו על ידי ה-HOC. אם היא נמצאת אני ממיר אותה למספר. אם לא, אני מחזיר 0 והכול נכנס לסטייט. זה הכול. אני בודק, כמובן, אם props.load קיימת.

אז למה להיכנס לכל הסרט הזה של ה-HOC ולא פשוט להכניס את ה-localStorage.set וה-localStorage.get לקומפוננטת CounterUp מההתחלה? למה לטרוח לעטוף אותה ב-HOC? התשובה היא שעכשיו אני יכול לקחת את ה-HOC הזה ולעטוף איתו קומפוננטות נוספות ולתת להן, כבמטה קסם, את היכולת לקרוא ולכתוב לתוך ה-localStorage בלי להתאמץ כלל ובלי לכתוב שוב ושוב localStorage.set ו-localStorage.get. וזה דבר חשוב מאוד, כי לפעמים יש לנו פונקציונליות שאנחנו צריכים להטמיע בקומפוננטה שלנו בנוגע לתקשורת לצד השרת וזה חוסך המון-המון קוד מיותר. הקריאות לצד השרת, או במקרה הזה ל-localStorage, תמיד יהיו מרוכזות במקום אחד.

תרגיל:

הוסיפו ל-HOC שכתבנו WithStorage – מתודת clear שמנקה את ה-localStorage.

רמז: ניקוי ה-localStorage נעשה באמצעות:

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

פתרון:

```
const WithStorage = (WrappedComponent: JSX.Element) => {
  function HOC(props: any): JSX.Element {
    function save(key: string, data: any): void {
      localStorage.setItem(key, data);
    }
    function load(key: string): any {
      return localStorage.getItem(key);
    }
    function clear(): void {
      localStorage.clear();
    }
    return (
      <WrappedComponent {...props} save={save} load={load} clear={clear}
/>
    );
  }
  return HOC;
};
export default WithStorage;
```

אין בעיה להוסיף כמה פונקציות שנרצה ל-HOC ולהעשיר בכמה props שבא לנו את הקומפוננטות שעוברות דרך ה-HOC. במקרה הזה פשוט הוספנו מתודה נוספת שנקראת clear ועושה את הפעולה הבאה:

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

תרגיל:

בקומפוננטת CounterUp שהודגמה בפרק ועטופה ב-HOC ששמו WithStorage, הוסיפו כפתור שמבצע איפוס של המונה.

רמז: איפוס המונה מתבצע באמצעות הפעלת מתודת clear שהוספנו אל WithStorage HOC בתרגיל הקודם. יש ליצור כפתור שיפעיל את המתודה הזו.

פתרון:

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


type propsInfo = {
  load:(value:string)=>void,
  save:(key: string, value:number)=>void,
  clear: ()=> void
}

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

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

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

  function resetHandler():void {
    setCount(0);
```

```
    if (props.clear) {
      props.clear();
    }
  }

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

export default CounterUp;
```

ראשית יש להוסיף כפתור. הכפתור הוא כפתור ה-reset:

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

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

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

מאיפה ה-props.clear מגיע? מה-HOC שעוטף את הקומפוננטה. זו הסיבה שאנו בודקים אם הוא קיים. איפה הוא עוטף את הקומפוננטה הזו? הראינו בפרק. בקומפוננטה Container:

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

const ComposedCompnent:number = WithStorage(CounterUp);

function MyContainer():JSX.Element {
    return (
        <ComposedCompnent />
    );
}

export default MyContainer;
```

offer@webit.com

פרק 19

ראוטינג

ראוטינג

עד כה עבדנו עם אפליקציות בנות דף אחד שבו יש קומפוננטות שונות. אבל זה כמובן לא ריאלי. באתר אמיתי, או באפליקציה אמיתית, יש כמה וכמה דפים. באתר אינטרנט בסגנון הישן, כל מעבר דף גרם לטעינה מחודשת של האתר כולו. בריאקט (וכן בכל אפליקציות הווב המודרניות) מעברי דפים נעשים ללא טעינה כלל מהשרת ובאמצעות קומפוננטות ראוטינג שמבצעות ניתוב של קומפוננטות דרך צד הלקוח, דבר המאפשר חוויית גלישה טובה בהרבה. הטכניקה הזו נקראת SPA – ראשי תיבות של Single Page Application.

ראוטינג (Routing), או בעברית תקנית "ניתוב", הוא מונח ידוע בכל הנוגע לאפליקציות ווביות, ומובן שיש אותו גם בריאקט. ריאקט, בניגוד לפריימוורקים אחרים (כמו אנגולר), לא קובעת איך לממש את הראוטינג, אלא יש כמה קומפוננטות ניווט חיצוניות שמומלצות על ידי ריאקט וכל אחד יכול לבחור מה שהוא רוצה. אפשר גם לממש את הראוטינג בעצמכם, אם כי זה לא מומלץ בהתחשב בעובדה שיש קומפוננטות רבות אחרות שכבר נבדקו במאות ובאלפי אתרים אחרים.
האובייקט בג'אווהסקריפט המכיל את הכתובת נקרא location. אפשר למצוא אותו תחת window.location ואנו יכולים לשנות אותו באמצעות ג'אווהסקריפט.

הבעיה היא ששינוי שלו גורם לטעינה מחדש וזה, כפי שכבר למדנו, גורם לרִנדור מחדש של כל הקומפוננטות ויוצר חוויית משתמש לא טובה, וגם עלול לגרור בעיית ביצועים. הפתרון? לשנות את ה-location בלי לטעון את הדף מחדש. יש שתי דרכים לעשות את זה: באמצעות שימוש באובייקט History, שנמצא תחת window.history, או באמצעות תוספת לכתובת בשם hash. השימוש ב-window.history נחשב למתקדם יותר והשימוש ב-hash נחשב לפשוט יותר. הבה נדגים ראוטינג באמצעות hash בלבד.

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

השרת אינו מסוגל "לראות" את מה שיש מאחורי הסימן # וכשאנו משנים אותו בדפדפן, הוא לא נשלח כלל לשרת. זו הסיבה ששינוי שלו, אפילו ידנית, לא יטען מחדש את הדף. בגלל זה # פופולרי מאוד לראוטינג ב-Single Page Application כמו באפליקציות ריאקט.

על מנת לתרגל ראוטינג ניצור 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.

קומפוננטת **Home.tsx**:

```
function Home():JSX.Element {
  return (
    <div><h1>Home</h1></div>
  );
}

export default Home;
```

קומפוננטת **About.tsx**:

```
function About():JSX.Element {
  return (
    <div><h1>About</h1></div>
  );
}

export default About;
```

קומפוננטת **Help.tsx**:

```
function Help():JSX.Element {
  return (
    <div><h1>Help</h1></div>
  );
}

export default Help;
```

כאמור, אלו קומפוננטות פשוטות לשם לימוד הראוטינג והבנתו בלבד.
כדי לבצע ראוטינג אנו יוצרים קומפוננטת אב שמכילה את סרגל הניווט ואת הקומפוננטה שתטען בכל פעם את קומפוננטת העמוד הפעילה, כלומר את Home, את About או את Help.

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

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

הפקודה הזו מורכבת משני חלקים – החלק הראשון:

```
window.location.hash
```

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

```
substr(1)
```

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

נבין את זרימת המידע בקומפוננטה לפני הכתיבה שלה:

1. בתחילה 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. מכניסים לסטייט את ה-route הפעיל.
4. השינוי בסטייט גורם לקומפוננטה להתרנדר מחדש. בִּרְנדור בודקים את ה-route הפעיל ובוחרים את הקומפוננטה.

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

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

function Router(): JSX.Element {
  const [route, setRoute] =
useState<string>(window.location.hash.substring(1));

  useEffect(() => {
    const handleHashChange = () => {
      setRoute(window.location.hash.substring(1));
    };

    window.addEventListener('hashchange', handleHashChange);

    return () => {
      window.removeEventListener('hashchange', handleHashChange);
    };
  }, []);

  let Child: React.ElementType;

  switch (route) {
    case '/about':
      Child = About;
      break;
    case '/help':
      Child = Help;
      break;
    default:
      Child = Home;
  }

  return (
    <div>
      <h1>App</h1>
      <ul>
        <li><a href="#/about">About</a></li>
        <li><a href="#/help">Help</a></li>
        <li><a href="#/home">Home</a></li>
```

```
      </ul>
      <Child />
    </div>
  );
}

export default Router;
```

את הקומפוננטה נציב כמובן ב-App באפליקציית Create React App שלנו:

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

function App():JSX.Element {
  return (
    <div className="App">
    <header className="App-header">
      <Router />
    </header>
    </div>
  );
}
export default App;
```

ננסה להבין מה קורה בקומפוננטת Router. הדבר הראשון שאני עושה הוא ליצור סטייט בשם route ולתת לו ערך ראשוני של מה שיש בכתובת #:

```
const [route, setRoute] =
useState<string>(window.location.hash.substring(1));
```

למשל, אם אני טוען את הקומפוננטה כשאני נמצא ב:

http://localhost:3000/#/home

הסטייט יהיה home/. הקוד יחזיר לי את השורה הבאה:

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

הדבר השני שאני עושה הוא שאני גורם לקומפוננטה, באמצעות ההוק useEffect, ברִנדור הראשוני בלבד, להצמיד אירוע שישנה את ה-route בסטייט בהתאם למה שיש אחרי ה-#. אני מוודא שמדובר ברִנדור הראשוני בלבד כי אני מעביר ב-useEffect פרמטר שני של מערך ריק:

```
useEffect(() => {
const handleHashChange = () => {
    setRoute(window.location.hash.substring(1));
};

window.addEventListener('hashchange', handleHashChange);

return () => {
    window.removeEventListener('hashchange', handleHashChange);
};
}, []);
```

שימו לב: מפני שהפרמטר השני שמועבר ל-useEffect הוא מערך ריק [], אז הפונקציה שאנו מעבירים תופעל רק ברִנדור הראשוני של הקומפוננטה – כלומר פעם אחת. זה חשוב, כי אם נצמיד אירוע ל-hashchange בכל רִנדור של הקומפוננטה הזו תהיה לנו זליגת זיכרון חמורה. אנו מצמידים אירוע פעם אחת בלבד.

הדבר השלישי בעצם הוא לבצע בדיקה מה הסטייט הפעיל ומחליפה את משתנה Child בקומפוננטה המתאימה לסטייט. שימו לב שאני משתמש במשתנה שמתחיל באות גדולה – Child ולא child. זה משום שמדובר במשתנה המכיל קומפוננטה ואני מציין בפני ריאקט שמדובר בקומפוננטה:

```
let Child: React.ElementType;

switch (route) {
case '/about':
    Child = About;
    break;
case '/help':
    Child = Help;
    break;
default:
    Child = Home;
}
```

חשוב: כדאי לשים לב שהמשתנים About, Help ו-Home מגיעים מהצהרת ה-import.

הצעד האחרון הוא לרנדר את התוצאה:

```
return (
<div>
    <h1>App</h1>
    <ul>
    <li><a href="#/about">About</a></li>
    <li><a href="#/help">Help</a></li>
    <li><a href="#/home">Home</a></li>
    </ul>
    <Child />
</div>
);
```

התוצאה מתרנדרת ברכיב Child, שמשתנה בכל פעם. המשתמש לוחץ על קישור, והאירוע של ה-hashchange עובד ומשנה את הסטייט. שינוי הסטייט גורם לרִנדור מחדש, שבו Child מקבל קומפוננטה חדשה.

כאמור, כך זה נראה באופן מאוד גס ופשוט. אם נרצה שזה יעבוד עם קישורים אמיתיים ולא עם hash נצטרך להשתמש ברכיבים שונים במקצת ולהפריד, כמובן, בין הקונפיגורציה לרכיב הראוטינג (למשל לשים את כל הנתיבים בקובץ JSON או על השרת), אבל בגדול – כך זה נראה. ראוטינג בריאקט אינו וודו וקל לממש אותו, אבל בעולם האמיתי נעדיף להשתמש בקומפוננטה מוכנה מראש.

הקומפוננטה הפופולרית ביותר נקראת, בפשטות, react-router, ואנו נלמד עליה בנוגע לראוטינג. אותם העקרונות המנחים אותנו רלוונטיים גם לראוטינג של כל קומפוננטה. הקומפוננטה הזו משתמשת ממש בכתובות אמיתיות ולא ב-#, אך העיקרון זהה. react-router היא ספריית ראוטינג כללית וספרייה נוספת שלה, react-router-dom, היא המשלימה לאפליקציות ווב. אם כך, נלמד להשתמש בשתיהן.

ראשית, ניכנס לדף של react-router:

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

אפשר לראות שיש שם מדריך מסודר ודוקומנטציה טובה. זה כלל מפתח בנוגע לכל קומפוננטה שאתם רוצים להשתמש בה בריאקט (ובכלל בכל מערכת) – דוקומנטציה טובה היא הכרחית. אם אין דוקומנטציה, עדיף לא להשתמש באותה קומפוננטה. אבל זה לא המקרה פה.

כיוון שמדובר בקומפוננטה חיצונית, אנו נתקין אותה. ההתקנה נעשית, בדיוק כמו כל קומפוננטה חיצונית אחרת, באמצעות npm שעליו למדנו בקצרה בפרק על קומפוננטות חיצוניות. נכנסים עם הטרמינל אל המיקום של הפרויקט שלנו ומתקינים את קומפוננטת react-router ואת react-router-dom באמצעות:

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

גם כאן, על מנת להדגים, נשתמש בקומפוננטות Home, About ו-Help. בניגוד לדוגמה שהראינו, של ראוטינג בסיסי, react-router-dom עובדת עם כתובות אינטרנט אמיתיות – כלומר ללא הסימן #. יש לכך כמה יתרונות והמימוש נעשה באותה הדרך.

המתודולוגיה זהה. אני מגדיר את הראוטינג ואיזו קומפוננטה נטענת בעקבותיו:

```
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():JSX.Element {
    return (
        <Router>
            <div>
                <h1>App</h1>
                <ul>
                    <li>
                        <Link to="/">Home</Link>
                    </li>
                    <li>
                        <Link to="/about">About</Link>
                    </li>
                    <li>
                        <Link to="/help">Help</Link>
                    </li>
                </ul>
                <hr />
                <Switch>
                    <Route exact path="/">
                        <Home />
```

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

export default MyRouter;
```

אם עברתם על תת-הפרק שמסביר איך מבצעים את הראוטינג ללא מודול חיצוני, זה יכול להיראות לכם פשוט עד כדי גיחוך. מאחורי הקומפוננטה יש שתי תפיסות עיקריות:

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

זה הכול. ברגע שמבינים איך ראוטינג עובד באופן טבעי, גם הקומפוננטות של הראוטינג נראות פשוטות.

קל מאוד להשתמש גם בפרמטרים של URL. בדוקומנטציה מוסבר איך לעשות זאת. כל מה שעלינו לעשות הוא להוסיף את הפרמטר עם נקודתיים בקומפוננטת Route שנמצאת בתוך ה-Routes:

```
import { Routes, Route, Link, Outlet, BrowserRouter } from "react-
router-dom";
import Home from "./Home";
import Help from "./Help";
import About from "./About";

function Layout(): JSX.Element {
  return (
    <div>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/help/123456">Help</Link>
          </li>
        </ul>
      </nav>
      <Outlet />
    </div>
  );
}
function MyRouter(): JSX.Element {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="about" element={<About />} />
          <Route path="help/:id" element={<Help />} />
        </Route>
      </Routes>
    </BrowserRouter>
```

```
  );
}
export default MyRouter;
```

השינוי היחיד שבוצע מהראוטינג הבסיסי יותר הוא:

```
<Route path="help/:id" element={<Help />} />
```

זה בעצם מאפשר לנו להכניס פרמטרים, למשל:

http://localhost:3000/help/6382020

והפרמטר הזה יהיה זמין לנו בקומפוננטת Help באמצעות useParams, שהיא פונקציה שמגיעה עם react-router. למשל כך:

```
import {
    useParams
  } from 'react-router-dom';


function Help():JSX.Element {
    let params = useParams();

    return (
      <div>
        <h1>Help</h1>
        <p>ID: {params.id}</p>
      </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 שכדאי לבחון. אבל כאמור, אם הבנתם איך מממשים ראוטינג בעצמכם ואיך העיקרון עובד, לא יהיה לכם קשה להבין ולהשתמש בקומפוננטות של ניווט.

תרגיל:

צרו שתי קומפוננטות: Home, שזהה לקומפוננטה מתחילת הפרק, והקומפוננטה הזו, המציגה תמונה אקראית מהרשת:

```
function NiceImage():JSX.Element {
  let imgSrc:string = 'https://picsum.photos/id/237/200/300';
  return (
    <img src={imgSrc} />
  );
}

export default NiceImage;
```

צרו אפליקציית ריאקט שמשתמשת בראוטינג. באמצעות הראוטינג אפשר להחליף בין Home לבין NiceImage. אל תשתמשו בראוטינג של קומפוננטה חיצונית.

פתרון:

```
import { useState, useEffect } from 'react';
import Home from './Home';
import NiceImage from './NiceImage';

function Router(): JSX.Element {
  const [route, setRoute] =
useState<string>(window.location.hash.substring(1));

  useEffect(() => {
    const handleHashChange = () => {
      setRoute(window.location.hash.substring(1));
    };

    window.addEventListener('hashchange', handleHashChange);

    return () => {
      window.removeEventListener('hashchange', handleHashChange);
    };
```

```
  }, []);

  let Child: React.ElementType;

  switch (route) {
    case '/niceimage':
      Child = NiceImage;
      break;
    default:
      Child = Home;
  }

  return (
    <div>
      <h1>App</h1>
      <ul>
        <li><a href="#/niceimage">Nice Image</a></li>
        <li><a href="#/home">Home</a></li>
      </ul>
      <Child />
    </div>
  );
}

export default Router;
```

זה שינוי קל מהדוגמה שהוצגה בפרק עצמו. מה שחשוב לראות הוא איך ה-switch case, שמופעל בכל פעם שה-hash (שזה 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.

פתרון:

```
import { Routes, Route, Link, Outlet, BrowserRouter } from "react-router-dom";
import Home from "./Home";
import NiceImage from './NiceImage';

function Layout(): JSX.Element {
  return (
    <div>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/nice-image">Nice Image</Link>
          </li>
        </ul>
      </nav>
      <Outlet />
    </div>
  );
}
function MyRouter(): JSX.Element {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="nice-image" element={<NiceImage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}
export default MyRouter;
```

כיוון שבמדובר בראוטינג פשוט, אני נדרש לעשות את הצעדים הבאים באפליקציה שלי:

1. התקנת react-router באמצעות: npm install react-router-dom.
2. יצירת רכיב שנקרא MyRouter (או כל שם אחר).
3. ביצוע import לכל הקומפוננטות שאני צריך.
4. עטיפת כל הקומפוננטה בקומפוננטה ראשית מסוג BrowserRouter.
5. תפריט וקישורים באמצעות רכיב Link:

```
<Link to="/nice-image">Nice Image</Link>
```

6. יצירת קומפוננטת Layout למקום שבו אמורות להיות הקומפוננטות שאני מגיע אליהן בניווט.
7. להציב בתוך קומפוננטת Routes את הקומפוננטות שנטענות כתוצאה מפעולת 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 ניווט באמצעות פרמטרים, כך שתוכלו להכניס מספר ב-URL, למשל:

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

או:

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

והתמונה עם ה-id המתאים תיטען.

פתרון:

קומפוננטת הראוטינג:

```
import { Routes, Route, Link, Outlet, BrowserRouter } from "react-
router-dom";
import Home from "./Home";
import NiceImage from './NiceImage';

function Layout(): JSX.Element {
  return (
    <div>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/nice-image/200">Nice Image 200</Link>
          </li>
          <li>
            <Link to="/nice-image/44">Nice Image 44</Link>
          </li>
        </ul>
      </nav>
      <Outlet />
    </div>
  );
}
function MyRouter(): JSX.Element {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="nice-image/:id" element={<NiceImage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}
export default MyRouter;
```

קומפוננטת NiceImage:

```
import {
    useParams
  } from 'react-router-dom';

function NiceImage():JSX.Element {
    let params = useParams();
    let imgSrc:string =
`https://picsum.photos/id/${params.id}/200/300`;
    return (
      <img src={imgSrc} />
    );
}
export default NiceImage;
```

בראוטינג הפתרון הוא פשוט ביותר – הוספת הפרמטר id ב-path בקומפוננטת Route.
בקומפוננטה שאמורה לתפוס את ה-id, הלוא היא NiceImage, צריך לבצע import ל-UseParams שמגיעה אלינו מ-react-router, לקבל את ה-id ולשתול אותו ב-imgSrc.
אפשר לממש גם שדה או תיבת Select שבהם כותבים מספר, והתמונה המתאימה תיטען.

otherworld.co.il

פרק 20

קונטקסט

קונטקסט

קונטקסט הוא מונח חשוב בריאקט. הוא מאפשר לנו לחלוק מידע בין קומפוננטות שונות, בדומה לסטייט, אבל הוא גלובלי. כיום אנו מכירים שתי דרכים להעברת מידע בקומפוננטות: האחת היא באמצעות props, כאשר קומפוננטת האב מעבירה לקומפוננטת הבנות מידע דרך ה-props, והשנייה היא אירועים – קומפוננטת הבת מפעילה אירוע כתוצאה מפעולה כלשהי (משתמש או טיימר). אם קומפוננטת האב העבירה פונקציה שמופעלת כתוצאה מאירוע כלשהו (Handler), הוא יופעל כאשר אירוע מופעל בקומפוננטת הבת.

הבעיה היא שזה יכול להסתבך. לפעמים קומפוננטות רבות מעצי היררכיות שונים צריכות לחלוק מידע, כמו למשל סטיילינג (עיצוב) – מידע על עיצוב כמו גודל פונט, פלטת צבעים וגדלים, או מידע על משתמש. קומפוננטה מקבלת מצד השרת את המידע על המשתמש וצריכה להעביר את המידע הזה הלאה. לפעמים יש מידע בקומפוננטת אב מסוימת וצריך להעביר אותו לקומפוננטה שנמצאת חמש רמות מתחתיה בהיררכיה ולשרשר את המידע. זה מורכב. קונטקסט מאפשר לנו לשמור מאגר מידע גדול שממנו כל קומפוננטה יכולה לקרוא, מעין סטייט גלובלי.

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 יצליח לגשת לקונטקסט שאותו הפרוביידר מספק. הפרוביידר יכול לאכלס את הקונטקסט באופן סטטי או לבצע קריאה ל-API חיצוני. הצרכן מקבל גישה לקונטקסט באמצעות הוקים.

פרוביידר לא חייב להיות קומפוננטה ובדרך כלל לא בנוי כקומפוננטה אלא כקובץ טייפסקריפט רגיל, למשל UserContext.ts, שמגדיר קונטקסט של משתמש:

```
import React from 'react';

export type UserInfo = {
  data: {
    name: string;
    surName: string;
  };
  changeUser: Function;
};

const initValue: UserInfo = {
  data: {
    surName: "",
    name: "",
  },
  changeUser: () => {},
};

const UserContext = React.createContext<UserInfo>(initValue);
export const UserProvider = UserContext.Provider;
export const UserConsumer = UserContext.Consumer;
export default UserContext;
```

כדאי לשים לב שאין לנו פה קומפוננטה, אלא יצירת קונטקסט באמצעות:

```
const UserContext = React.createContext<UserInfo>(initValue);
```

האובייקט initValue, שמועבר כפרמטר ל-createContext, הוא בעצם ברירת מחדל – ערך שמוחזר רק כאשר קומפוננטה מבצעת קריאה לפרוביידר הזה, אם הוא לא משויך אליה
שימו לב לטייפ פה – אני משתמש בטייפ UserInfo ומעביר אותו באופן גנרי ל createContext.

במקרה הזה אני מגדיר את UserInfo בתוך הקובץ עצמו כדי לפשט את הדוגמה. במציאות אני אגדיר את סוג המידע הזה בקובץ נפרד ואעשה לו export כי אני צריך להשתמש בו בעוד שני מקומות – בקומפוננטה הצורכת את הקונטקסט.

אני משתמש ב-API של קונטקסט כדי ליצור ולייצא את הפרוביידר, את הצרכן ואת הקונטקסט. אני יכול לבחור לפרוביידר, לצרכן או לקונטקסט תחילית כרצוני. במקרה הזה, כיוון שמדובר בקונטקסט של משתמש, בחרתי בתחילית של User.

אחרי שהגדרתי קונטקסט, אני יכול להשתמש בפרוביידר. הפרוביידר מספק לכל הקומפוננטות שעוטפות אותו את הקונטקסט שהוא נותן. אני יכול לשים אותו בכל מקום, אפילו ב-App. הינה דוגמה:

```
import './App.css';
import { UserProvider } from './userContext'
import Welcome from './Welcome';

type userInfo = {
  name: string,
  surName: string,
  city: string,
}

function App():JSX.Element {
  const user:userInfo = {
    name: 'Ran',
    surName: 'Bar-Zik',
    city: 'Petah-Tiqwa'
  };
  return (
    <div className="App">
    <header className="App-header">
    <UserProvider value={user}>
      <Welcome />
    </UserProvider>
    </header>
    </div>
  );
}
export default App;
```

ראשית אני מייבא את הפרוביידר מהקונטקסט שיצרתי. כזכור, אני מייצא משם פרוביידר, צרכן וקונטקסט.

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

אני בוחר את המידע שאני רוצה להעביר בקונטקסט. המידע יכול להיות סטטי או מידע שמגיע משירות (כלומר מהשרת). בדוגמה הזו אני משתמש במידע סטטי:

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

שימו לב שאני חייב להגדיר כאן סוג מידע בשם userInfo או בכל שם אחר, אבל אני לא יכול להגדיר משתנים בלי סוג מידע. אנו עדיין בטייפסקריפט

הצעד האחרון הוא להשתמש בפרוביידר כמו בקומפוננטה רגילה ולהעביר אליו את המידע. כל מה שיהיה עטוף בקומפוננטה הזו וכמובן כל הקומפוננטות שמתחת לקומפוננטה העטופה, כלומר אלו שתחתיה בהיררכיה, יקבלו גישה לקונטקסט:

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

עכשיו במקום Any Component נשים קומפוננטה אמיתית, למשל Welcome.tsx:

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

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

איך הקומפוננטה הזו תיראה?

```
import { useContext } from 'react'
import UserContext from './userContext'

type userInfo = {
  name: string,
  surName: string
}

function Welcome():JSX.Element {
  const user:userInfo = useContext<userInfo>(UserContext);
  return (
    <span>Hello, {user.name} {user.surName}!</span>
  );
}
export default Welcome;
```

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

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

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

```
import UserContext from './UserContext'
```

השלב הבא והאחרון הוא פשוט לצרוך את המידע שהפרוביידר הזה מביא לנו. ללא טייפסקריפט, הפונקציה הזו נראית כך:

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

אבל עם טייפסקריפט זה נראה כך, זה מאיים בהתחלה אבל נפרק את זה:

```
  const user:userInfo = useContext<userInfo>(UserContext);
```

ראשית הגדרת הקבוע עם סוג המידע userInfo. את זה אנו אמורים להכיר:

```
const user:userInfo
```

השלב השני הוא העברת סוג המידע הזה ל userContext באמצעות גנריות. כלומר אנו מגדירים ל userContext את סוג המידע שהיא צריכה להחזיר.

```
(UserContext as React.Context<userInfo>);
```

אנו חייבים להגדיר את הסוג הזה כ React.Context. אבל ReactContext זה לא סוג מידע רגיל, זה סוג מידע שבעצמו מקבל את ההגדרה באמצעות גנריות ואני נדרש להעביר לו את סוג המידע. userInfo.

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

אם הפרוביידר מתעדכן מסיבה מסוימת – הוא ירנדר מייד את הקומפוננטות שמשתמשות בו. שימו לב שהקומפוננטה צריכה להיות מודעת ל-UserContext ולעשות לו import.

אבל מה קורה אם אנו רוצים לעדכן את הקונטקסט ממקום אחר שהוא לא הפרוביידר? אם אנו רוצים לתת לפרוביידר אפשרות להתעדכן על ידי אחת מהקומפוננטות בתחתית ההיררכיה, אנו צריכים להעביר, נוסף על הקונטקסט, גם פונקציה שתשנה אותו.

אני אדגים זאת באמצעות הוספת כפתור טעינת משתמש בקומפוננטת Welcome שטוען משתמש חדש.

הצעד הראשון שצריך לעשות הוא ליצור סטייט בפונקציה המכילה את הפרוביידר. במקרה הזה App.tsx. בסטייט הזה אני מכניס את הערכים הראשוניים שלי ופונקציה המשנה את אותו סטייט. לפרוביידר אני כבר לא מכניס את אובייקט המידע בלבד אלא גם את אובייקט המידע תחת תכונת data וגם את הפונקציה המשנה את הסטייט. אני יכול לקרוא לפונקציה המשנה את הסטייט באיזה שם שאני רוצה. כאן בחרתי ב changeUser.

```
import './App.css';
import { useState } from "react";
import { UserProvider } from './userContext'
import Welcome from './Welcome';

type UserInfo = {
  name: string,
  surName: string,
}

function App():JSX.Element {

  const [user, setUser] = useState({
    name: "Ran",
    surName: "Bar-Zik",
  });
  const providerOptions = {
    data: user,
    changeUser: (value: UserInfo) => setUser(value),
  };
```

```
  return (
    <div className="App">
    <header className="App-header">
    <UserProvider value={providerOptions}>
      <Welcome />
    </UserProvider>
    </header>
    </div>
  );
}
export default App;
```

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

בפרוביידר נמצא data ושם הפונקציה המשנה אותו. אני צורך אותם באמצעות UserContext.

אם Welcome.tsx רוצה לשנות את המידע הזה, הוא צריך להפעיל את הפונקציה שמשנה את הסטייט שבו נמצא הפרוביידר, וכך הוא ישנה אותו. כך פונקצית Welcome נראה.

```
import { useContext } from "react";
import UserContext from "./userContext";
import Button from "@mui/material/Button";

type UserInfo = {
  name: string;
  surName: string;
};

function Welcome(): JSX.Element {
  const { data, changeUser } = useContext(UserContext);

  const newUser: UserInfo = {
    name: "Moshe",
    surName: "Cohen",
  };

  function clickHandler(): void {
```

```
    changeUser(newUser);
  }

  return (
    <div>
      <span>
        Hello, {data.name} {data.surName}!
      </span>
      <Button variant="contained" color="primary"
onClick={clickHandler}>
        Load another user
      </Button>
    </div>
  );
}

export default Welcome;
```

בקומפוננטת Welcome.tsx, ראשית אני שואב את אובייקט המשתמש (כדי שאוכל להציג אותו) וכן את הפונקציה שמשנה את הסטייט של הפרוביידר. את הפונקציה אני מפעיל באירוע הלחיצה של הכפתור.

אז בעצם מה שקורה הוא:

1. אני מגדיר קונטקסט בקובץ נפרד.
2. בוחר היכן להציב את הפרוביידר שלי, בדרך כלל בקומפוננטת אב. במקום שבו אני מציב את הפרוביידר, אני מזריק ערך. הדרך הכי טובה לעשות את זה היא באמצעות סטייט, אם כי זו לא חובה.
3. אני יכול להעביר יחד עם הערך פונקציה שמשנה את הסטייט של הפרוביידר. כל מי שמשתמש בקונטקסט הזה יכול (אם הוא רוצה) להשתמש בה ולשנות את הקונטקסט לכולם.

קונטקסט הוא דרך מצוינת לחלוק מידע בין קומפוננטות שונות וגם קל להשתמש בו עם הוקים. הדרך לעדכון קונטקסט יכולה להיראות מסורבלת, אבל אם מבינים שברגע שיוצרים קונטקסט ומידע גם יוצרים פונקציה שמשנה אותו ומעבירים את שניהם – אז הכול נהיה פשוט.

תרגיל:

העתיקו את קומפוננטת Welcome.tsx וקומפוננטת App.tsx ל-Create React App שלכם.

הוסיפו את שם העיר לאובייקט המידע. הציגו את שם העיר בקומפוננטה נפרדת בשם CityName.tsx שמציגה את שם העיר של המשתמש. הקומפוננטה תוצב בקומפוננטת Welcome ותצרוך ישירות מהפרוביידר.

פתרון:

ראשית, ניצור את הקומפוננטה. הקומפוננטה היא קומפוננטה שאמורה לצרוך קונטקסט ונראית כך:

```
import { useContext } from "react";
import UserContext from "./UserContext";

type UserInfo = {
  name: string;
  surName: string;
  city: string;
};
function CityName(): JSX.Element {
  const user: UserInfo = useContext<UserInfo>(UserContext);
  return <span>You are from {user.city} </span>;
}
export default CityName;
```

השורות המעניינות מבחינתנו הן:

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

כעת נייבא את ההוק useContext:

```
import UserContext from "./UserContext";
```

הקומפוננטה הזו דומה לקומפוננטת Welcome.tsx.

ומה קורה בקומפוננטת Welcome? אין חדש. אנחנו רק צריכים להציב את קומפוננטת CityName.tsx כדי להציג את שם העיר.

```
import { useContext } from 'react'
import UserContext from './UserContext'
import CityName from './CityName';

type UserInfo = {
  name: string,
  surName: string,
  city: string,
}

function Welcome():JSX.Element {
  const user:UserInfo = useContext<UserInfo>
    (UserContext);
  return (
    <div>
        <span>Hello, {user.name} {user.surName}!</span>
        <span><CityName /></span>
    </div>
  );
}
export default Welcome;
```

האם צריך לקשר את קומפוננת CityName לפרוביידר? לא. כל עוד היא בת, נכדה, נינה או הבת של הנינה של ה-UserProvider זה לא משנה – היא יכולה להשתמש בקונטקסט.

offer@webit.com

פרק 21

חיבור לשרת עם סרוויסים

חיבור לשרת עם סרוויסים

פרק זה מניח ידע מוקדם ב-AJAX ובהבדל בין הבקשות השונות – לפחות ברמת פונקציית ה-fetch הטבעית.

חיבור לצד שרת הוא קריטי 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 לקונטקסט. את הקריאות אנו מבצעים באמצעות AJAX ואפשר להשתמש ב-fetch שמגיע עם ג'אווהסקריפט או בכל ספרייה אחרת, כמו הספרייה הנפוצה Axios.

כשאנו מריצים קוד בריאקט, הוא רץ בדפדפן בלבד. נכון, Create React App מרימה שרת פיתוח במחשב שלנו לשם הנוחות. אבל בסופו של יום, בסביבת הפרודקשן, הסביבה האמיתית, השרת יכול להיות שונה לחלוטין. הדפדפן יכול לקבל את קובצי הג'אווהסקריפט שמורכבים מספריית ריאקט ומהקוד שלכם מכל שרת שהוא. אחרי שדפדפן הלקוח מוריד את קובצי הג'אווהסקריפט האלו, הוא מריץ את האפליקציה הבנויה על ידי ריאקט על הדפדפן שלו. אם הוא צריך מידע מהשרת, הקוד שלכם אמור להביא לו אותו.

בדרך כלל, מתכנתי צד השרת כותבים API בצד השרת כדי שיחזיר לכם מידע, והאחריות של מתכנתי ריאקט היא לשלוח בקשות לשרת. כך, למשל, אתם מממשים בראוטינג דף התחברות עם שם משתמש וסיסמה. המשתמש מקליד שם משתמש וסיסמה ולוחץ על כפתור "שלח". האחריות שלכם היא לשלוח, באמצעות AJAX, את שם המשתמש והסיסמה ל-API של צד השרת. צד השרת אמור להחזיר לכם תשובה של 200 ואובייקט המכיל את פרטי המשתמש, בהנחה שהכול מצליח, ולשתול עוגייה בצד המשתמש. האחריות שלכם היא לקחת את המידע הזה ולהכניס אותו לקונטקסט כדי שכל המידע באתר יוצג למשתמש המחובר כמו שצריך.

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

יש כמה דרכים לנהל את הבקשות – אפשר לכתוב אותן בתוך הקומפוננטה ממש. אנו נתאמן עם ה-API של בדיחות צ'אק נוריס, שבעצם מאפשר לנו לקבל בדיחת צ'אק נוריס בקישור הבא:

```
https://api.chucknorris.io/jokes/random
```

נסו בעצמכם! העתיקו והדביקו את הקישור בדפדפן וצפו ב-JSON שמגיע.

איך נממש את זה? כאמור – אפשר בקומפוננטה:

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

function Welcome():JSX.Element {
  const [joke, setJoke] = useState<string>('Joke is loading...');

  function getJoke():void {
    fetch('https://api.chucknorris.io/jokes/random', {})
    .then((response) => {
        return response.json();
    })
    .then((jsonObject) => {
      setJoke(jsonObject.value);
    });
  }

  useEffect(getJoke , [])

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

export default Welcome;
```

אנו מכניסים את קריאת ה-AJAX, באמצעות fetch, אל פונקציה מסודרת שאנו מפעילים רק פעם אחת, באמצעות ההוק useEffect שמקבל כפרמטר ראשון את הפונקציה להפעלה וכפרמטר שני מערך ריק, שבעצם אומר שהוא יופעל רק כאשר הקומפוננטה מרונדרת. הפונקציה שעושה fetch גם מכניסה את הבדיחה המתקבלת אל הסטייט.

הקוד הזה, שאמור להיות מובן מאוד אם אתם כבר מכירים AJAX ויודעים איך fetch עובד, הוא קוד ולידי, אבל הוא רעיון גרוע מאוד אם האפליקציה שלכם מעט יותר מורכבת, למשל אם יש דרישות נוספות כמו להכניס את תוצאות הבדיחה לקונטקסט, כדי שעוד קומפוננטות יוכלו להציג אותה, או לשנות את ה-API. זו הסיבה שמקובל מאוד להכניס קריאות לשרת לפונקציות ג'אווהסקריפט טהורות שנקראות סרוויסים (Services). יוצרים בפרויקט תיקייה שנקראת Services ושם שמים את כל הפונקציות המטפלות בקריאות לשרתים.

למשל, אם אני רוצה להמיר את הקריאה לקומפוננטה הזו בקריאה לשירות אמיתי, השירות ייראה כך:

```
export default function getChukJoke():object {
    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 הזה.

```
import { useState, useEffect } from 'react'
import getChukJoke from './services/chuckJokes';

function Welcome():JSX.Element {
  const [joke, setJoke] = useState<string> ('Joke is loading...');

  useEffect(getJoke , []);

  function getJoke():void {
    getChukJoke().then((joke) => {setJoke(joke);});
  }

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

export default Welcome;
```

יש שירותים שנבנים כקלאסים ויש כאלו שנבנים כפונקציות. כל דרך היא טובה – אבל כדאי לשים לב שלא משנה איזו דרך נבחרה – מדובר בטייפסקריפט טהור. ריאקט לא מתערבת בענייני הסרוויסים וכל אחד יכול לממש אותם לפי הבנתו.

סביר להניח שכאשר תעבדו מול מערכות אמיתיות, מי שיגדיר לכם איך הסרוויסים מתנהלים מול השרת יהיו מתכנתי הבק אנד, והם אלו שיגדירו לכם איך להתחבר.

תרגיל:

נדרשתם להציג בדיחת אבא רנדומלית באתר. מתכנת צד השרת הגדיר לכם API באופן הבא: יש לשלוח בקשת GET אל הכתובת:
https://icanhazdadjoke.com

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

רמז: בקשת fetch עושים באופן הבא:

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

פתרון:

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

משתמשים ב-service הזה כך:

```
import { useState, useEffect } from 'react'
import getDadJoke from './services/dadJokes';

function DadJoke():JSX.Element {
  const [joke, setJoke] = useState<string>('Joke is loading...');

  useEffect(getJoke , []);

  function getJoke():void {
    getDadJoke().then((joke) => {setJoke(joke);});
  }

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

export default DadJoke;
```

בגדול, זה לא ריאקט אלא ג'אווהסקריפט. הסרוויס הוא פונקציה שמייצרת בקשת AJAX. הבקשה נשלחת עם header ומחזירה אובייקט JSON. אנו לוקחים את האובייקט ומחזירים את ה-joke מתוכו.

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

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

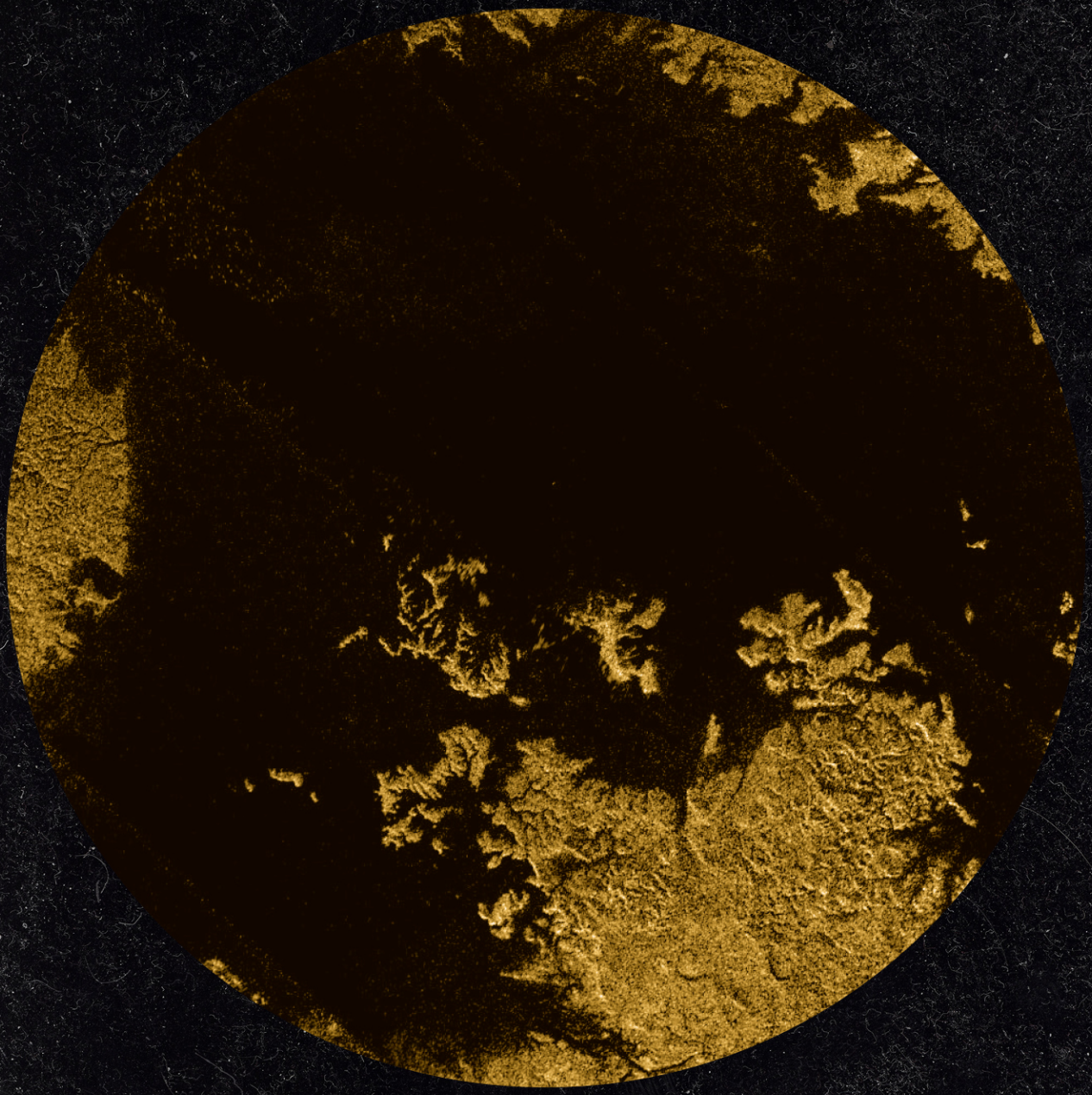
```
import { useState, useEffect } from 'react'
import getDadJoke from './services/dadJokes';

function DadJoke():JSX.Element {
  const [joke, setJoke] = useState<string>('Joke is loading...');

  useEffect(getJoke, []);

  function getJoke():void {
    getDadJoke()
      .then(
        (joke) => { setJoke(joke); },
        () => { setJoke('Sorry! Error!'); }
      );
  }

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

export default DadJoke;
```

או כמובן ניהול מסודר יותר. חשוב מאוד, בקריאות AJAX, לחשוב על מה שקורה אם השירות לא מחזיר את התשובה המצופה.

otherworldbits.com

פרק 22

מבוא לבדיקות עם JEST



מבוא לבדיקות עם jest

בדיקות אוטומטיות הן חלק חשוב מפיתוח מקצועי בריאקט. בפיתוח יש לנו כמה סוגי בדיקות: בדיקות יחידה (הבודקות את הקומפוננטה), בדיקות פונקציונליות (כדי לראות איך הקומפוננטה עובדת), בדיקות אינטגרציה (הבודקות איך הקומפוננטות השונות מתקשרות עם צד השרת) ובדיקות End to End (E2E) שבודקות את כל האפליקציה מקצה לקצה.

הבדיקות חיוניות לפיתוח נכון, וברוב החברות הטובות נהוג לכתוב בדיקות יחידה. הבדיקות האלו קריטיות כיוון שאם מתכנת אחר יעשה שינוי בקומפוננטה שלכם, הבדיקה תישבר והוא יבין שיש בעיה עם תסריט מסוים שבניתם עליו ובדקתם אותו. זו הסיבה שבדיקות הן דבר חשוב ולא מיועדות רק לאנשי QA. בעולם הפיתוח מקובל שמפתחים כותבים בדיקות יחידה.

בדיקות אוטומטיות הן סקריפטים, הכתובים בג'אווהסקריפט, ויש לכל בדיקה קריטריונים להצלחה. למשל: "אם לוחצים על כפתור, האפליקציה קוראת לשירות". הבדיקות האלו רצות באופן נפרד באמצעות פקודה מסוימת ומריצים אותן לפני כל הכנסת קוד חדש. בגלל זה בדיקות אוטומטיות הן חלק ממה שנקרא Continuous Integration, שהוא תהליך שקוד חדש עובר כשהוא משולב בקוד שנכתב קודם לכן. אם מישהו כתב תוספת לקוד שלנו (למשל הוסיף props נוספים) ויש לנו בדיקות טובות שרצות על הקוד החדש וכל הבדיקות רצות ואף אחת מהן לא נכשלת – אנו יודעים שכנראה נכתב קוד איכותי (או לפחות כזה שעבר את הבדיקות) ושנוכל להעלות אותו לאוויר.

ספריית התוכנה שמריצה את הבדיקות בריאקט נקראת jest. היא כבר מגיעה עם Create React App באופן מובנה. משתמשים ב-jest במקומות נוספים למעט ריאקט והיא ספרייה פופולרית מאוד לבדיקות. בבדיקות מקובל שקובץ הבדיקות נושא את השם של הקובץ המקורי עם התוספת spec או test. למשל, קובץ הבדיקות של Welcome.tsx יהיה Welcome.spec.js, אבל יש קונבנציות נוספות.

לא חייבים לכתוב את הבדיקות בטייפסקריפט וזה נחשב בחלק מהצוותים כעודף תקורה וניתן לכתוב את הבדיקות בג'אווהסקריפט בלבד.

הבה נכתוב בדיקה אוטומטית ראשונה עבור Welcome.tsx, שתיראה כך:

```
import React from 'react'

function Welcome():JSX.Element {
  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 שמפעיל את הקומפוננטה ובוחן 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.ts.

בתוכו אנו עושים import לריאקט ולרכיב שנקרא react-dom, שיסייע לנו לרנדר את הקומפוננטה. אנו נבדוק את תכונת ה-innerHTML של הקומפוננטה כדי לראות שהיא הדפיסה את הפלט כמו שצריך. כך עושים את זה:

```
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 חלקי הבדיקה.

חלק ראשון – import

אנו מייבאים את ריאקט (היא נדרשת על מנת שנוכל לרנדר קומפוננטות) ואת react-dom, וכמובן את הקומפוננטה שלנו. אלו החלקים הכי חשובים לבדיקה.

חלק שני – כתיבת מסגרת הבדיקה

it הוא חלק חשוב מאוד בבדיקות והוא בא כאובייקט גלובלי מ-jest. הוא מקבל שני ארגומנטים: האחד הוא שם הבדיקה והשני הוא פונקציית חץ שבתוכה הבדיקה.

חלק שלישי – הרצת הקומפוננטה

הצעד הראשון הוא ליצור HTML שבתוכו הקומפוננטה תתרנדר. מדובר בג'אווהסקריפט טהורה שבה יוצרים div סתמי ומקבלים רפרנס אליו:

```
const div = document.createElement('div');
```

הצעד השני הוא להשתמש ב-react-dom על מנת לרנדר את הקומפוננטה. הפונקציה ReactDOM.render מקבלת שני ארגומנטים: האחד הוא הקומפוננטה, שלה עשינו import, והשני הוא המקום שבו היא מתרנדרת, במקרה הזה ה-div שיצרנו.

חלק רביעי – הבדיקה

אחרי שהקומפוננטה רונדרה, אנו יכולים לבחון את ה-div שבתוכו היא נמצאת. אנו עושים את הבחינה באמצעות פונקציית expect. מדובר בפונקציה שגם היא באה עם jest והיא מקבלת ביטוי שאליו יש לשרשר את ההנחה. במקרה זה הביטוי הוא div.innerHTML וההנחה היא isEqual – 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 הזה 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 מריצים בטרמינל שכבר למדנו לעבוד איתו בפרקים קודמים. אם אנו עובדים עם Visual Studio Code, נפתח חלונית של טרמינל באמצעות בחירה בלשונית Terminal בחלק העליון של התוכנה ואז ב-New Terminal.

לחלופין, אתם יכולים לפתוח cmd ולנווט אל תיקיית הפרויקט.
לא משנה באיזו דרך בחרתם, על מנת להריץ את כל הבדיקות יש להקליד: npm test. מייד ירוץ התהליך ואם הבדיקה תעבור בהצלחה, תראו אישור.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PASS  src/App.test.js
PASS  src/Welcome.spec.js

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.289s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

אם הבדיקה לא תעבור בהצלחה, תוכלו לראות את הסיבה שבגללה 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 מדפיסה !Hello world ב-w קטנה, הנחתי שהיא מדפיסה !Hello World ב-W גדולה. רואים את השוני.

בדיקות לקומפוננטה עם props

אם לקומפוננטה יש props, עלינו להעביר אותם בבדיקה. זה קל יותר ממה שזה נשמע. הבה נבחן את הקומפוננטה הזו:

```
function Welcome(props: {name:string, surName:string}):JSX.Element {

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

export default Welcome;
```

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

```
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 הזו:

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

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

ברוב הפעמים נשתמש בספריית עזר לשם הבדיקות, כמו ספריית העזר Enzyme שמסייעת מאוד בבדיקת אירועים וברִנדור של הקומפוננטות והקומפוננטות הבנות (היא לא נלמדת במסגרת הספר).

פונקציית בדיקה – assert

בבדיקה שעשינו לשם הדוגמה, השתמשנו ב-toEqual. כלומר, 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 הזו נקראת פונקציית assert, וספריית jest מציעה לנו לא מעט פונקציות בדיקה כאלו.

not

כשאנו מוסיפים את הפונקציה not לפני פונקציית ההשוואה, אנו בעצם הופכים אותה. הבדיקה הזאת, למשל, בודקת שהטקסט הוא לא Error:

```
expect(div.innerHTML).not.toEqual('Error');
```

toBeTruthy

פונקציה שבודקת שיש לנו ערך שהוא לא undefined. זה חשוב כאשר לא אכפת לנו איזה ערך אנו מקבלים, העיקר שיהיה מדובר בערך אמיתי. בדרך כלל זה נעשה בבדיקות ראשוניות, כאשר אנו רק רוצים לוודא רִנדור של קומפוננטה או ערך מסוים. למשל, אם אני רוצה לדעת שיש גובה לקומפוננטה, ולא אכפת לי איזה גובה.

```
expect(div.clientHeight).toBeTruthy();
```

toBeFalsy

פונקציה שבודקת שלא קיבלנו ערך. היא שימושית אם רוצים לדעת שבמקרה הזה הבדיקה מחזירה לנו משהו שלא קיים. למשל, אם אני רוצה לוודא שאין id לקומפוננטה – כיוון שכמה קומפוננטות עם אותו id יכולות לעורר בעיות ב-DOM – אני אבצע את הבדיקה באופן הזה:

```
expect(div.id).toBeFalsy();
```

toContain

פונקציה שבודקת שהפרמטר שאנו מעבירים לה נמצא בטקסט ההשוואה. זה שימושי כאשר אנו לא רוצים לבצע בדיקה מלאה. אם נסתכל על הדוגמה שלנו, מדובר למשל בבדיקה שהשם Moshe מופיע:

```
expect(div.innerHTML).toContain('Moshe');
```

תרגיל:

נתונה הקומפוננטה Greeting. כתבו לה בדיקה שבודקת שהיא מדפיסה את הפלט Good Morning Moshe! כשמעבירים לה את ה-name המתאים.

```
function Greeting(props :{name: string}) :JSX.Element {

  return (
    <p>Good Morning {props.name}!</p>
  );
}

export default Greeting;
```

פתרון:

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

בדיוק כמו בדוגמה שהובאה בפרק, ייבאנו את הקומפוננטה ובאמצעות ReactDOM.render ייצרנו אותה בתוך ה-div. הבדיקה האמיתית היא השוואת הפלט של הקומפוננטה באמצעות div.innerHTML. את ההרצה 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.

פתרון:

```
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 הזה בבדיקה של toContain, כיוון שמה שמעניין אותנו הוא המילה הזו ולא כלל הבדיקה. בדרך כלל נשתמש ב-toContain, כיוון שאם נשנה את המבנה הפנימי של הקומפוננטה לא נשבור את הבדיקה.

תרגיל:

נתונה הקומפוננטה Random.tsx שמדפיסה מספר רנדומלי:

```
function Random() {

  return (
    <React.Fragment>{Math.random()}</React.Fragment>
  );
}

export default Random;
```

כתבו בדיקה שבודקת שהקומפוננטה מוציאה פלט מסוים.

פתרון:

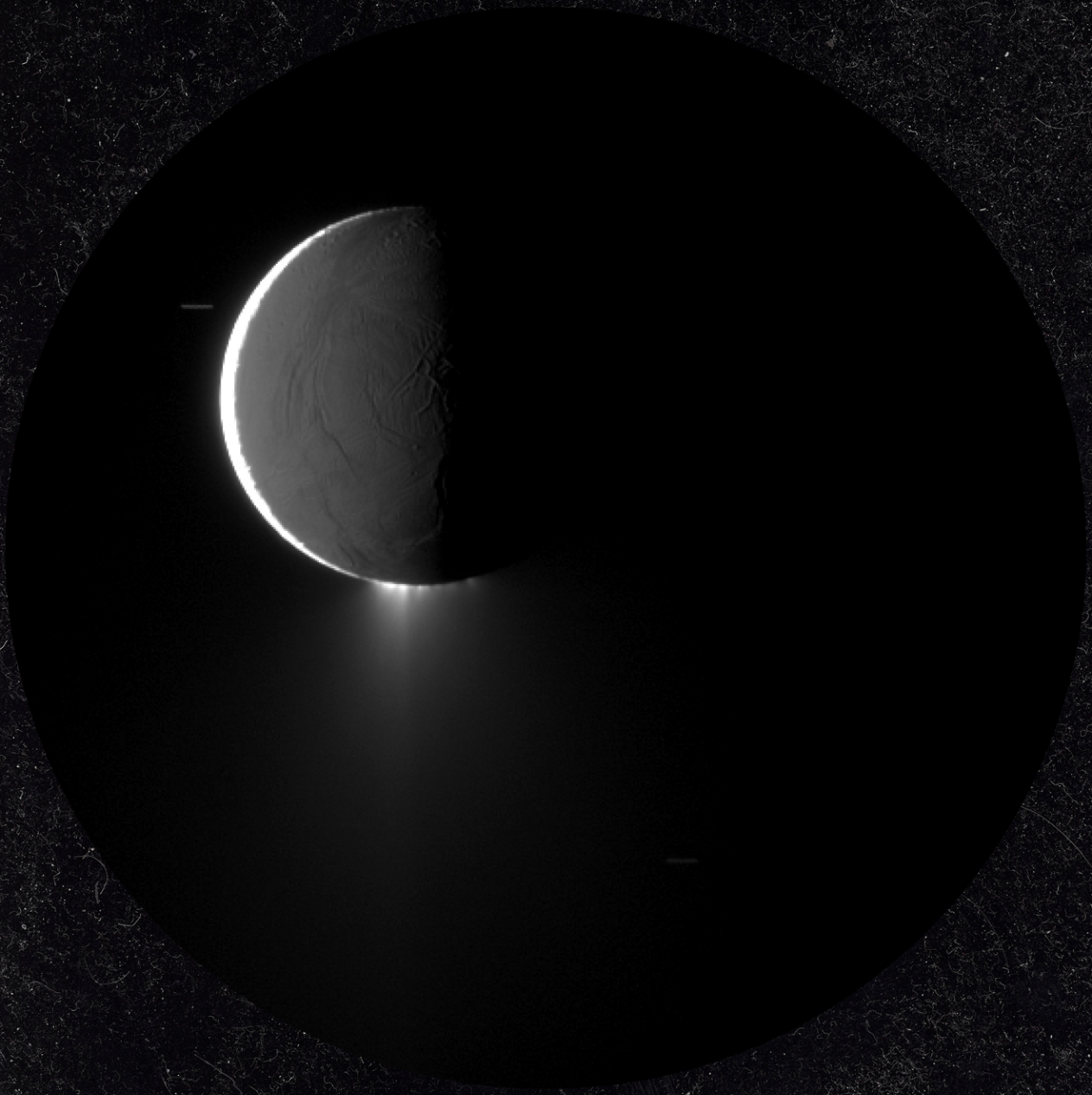
```
import ReactDOM from 'react-dom';
import Random from './Random';

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

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

offerrow.wix.com

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 חיצוניות, כתבנו שירותים ופונקציות – הגיעה העת להעלות אותה אל השרת. צריך לזכור שבסופו של יום כל האפליקציה היא קובץ ג'אווהסקריפט, קובץ HTML וקובצי CSS שנטענים על ידי הדפדפן. בקובץ הג'אווהסקריפט הזה, הדחוס והמחובר, יש הכול: ספריית ריאקט, babel, הקומפוננטות החיצוניות והקוד שכתבתם. הכול בקובץ אחד. הדפדפן טוען אותו והכול מתחיל להיבנות משם. יש כאלו שבוחרים לחלק את הקובץ הדחוס לכמה חלקים – למשל תלויות חיצוניות (ספריות צד שלישי כמו למשל React Material UI שעליה למדנו באחד הפרקים הקודמים) והאפליקציה שכתבנו. אבל בסופו של הבילד, התוצאה היא קבצים דחוסים.

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

באתר אמיתי אנו צריכים שרת אמיתי שיגיש את הקבצים לדפדפן. כשאני אומר "יגיש" אני מתכוון לתהליך שבמסגרתו אנו מקלידים כתובת של אתר אינטרנט בדפדפן. הדפדפן ניגש לשרת האתר והוא מחזיר לו קבצים שהדפדפן מציג.

כך למשל כשאני גולש לגוגל, אני מקליד בדפדפן google.com, השרת של גוגל מעביר לדפדפן 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 לדפדפן קבצים.
אילו קבצים?

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

איך אנו יוצרים את השרת הזה? יש הרבה דרכים. הדרך הפשוטה ביותר היא לשכור shared hosting באתר כלשהו ופשוט להציב עליו את קובץ ה-index.html וקובצי ה-CSS והג'אווהסקריפט (זה עובד אם עושים ראוטינג מבוסס #), באמצעות העתקה פשוטה, בתיקייה הראשית של השרת בעזרת הממשק של ניהול השרת שספק השרת מעניק לכם. הקלדת כתובת שם המתחם וכניסה אל האתר תגרום לטעינת קובץ ה-HTML וכמובן כל הקבצים המקושרים אליו.

יש דרכים רבות אחרות – לא חסרות שפות צד שרת שיש בהן שרתים. הפופולרית ביותר היא Node.js, אך יש גם את ג'אווה, #C ופייתון שמאפשרות יצירת שרתים. אם אתם, או מתכנת צד השרת בחברה שלכם, יוצרים שרת כזה – בסופו של דבר תצטרכו לגרום לו להחזיר ללקוח את דף ה-HTML והקבצים. הלקוח ייכנס, הקבצים ייטענו ומפה תהפוך האפליקציה לפעילה בדיוק כמו ב-Create React App.

איך מייצרים את קובץ ה-HTML, קובץ ה-CSS וכמובן קובצי הג'אווהסקריפט? באמצעות פקודת npm מיוחדת שיש להקיש בטרמינל – ממשק הפקודה הטקסטואלי. אם אתם משתמשים ב-Visual Studio Code, לחצו על Terminal בתפריט העליון ואז על New Terminal. ייפתח לכם חלון בתחתית המסך עם שורת פקודה טקסטואלית.

אם אין לכם Visual Studio Code, היכנסו אל ה-cmd בחלונות ונווטו אל התיקייה של האפליקציה.
מייד כשאתם יכולים להקליד בטרמינל, הקלידו:

```
npm run build
```

ירוץ תהליך שמאחד ומכווץ את קובצי הג'אווהסקריפט וה-CSS ויוצר קובץ index.html וגם קבצים סטטיים נוספים שמסייעים לשרת. כל הקבצים האלו ייכנסו אל תיקיית build – זו התיקייה שיש להעלות אל השרת כדי להשתמש באפליקציה שלכם, בין שמדובר בשרת של חברה גדולה ובין שמדובר ב-shared hosting:

```
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, אנגולר ודומותיהן. התהליך הזה מתרחש אוטומטית כאשר מתכנת מוסיף קוד לפרויקט והתהליך משולב בדרך כלל עם הבדיקות האוטומטיות.

אפשר לבדוק את הבילד באמצעות מודול npm שנקרא serve. מתקינים אותו גלובלית באמצעות הקלדת הפקודה הזו בטרמינל:

```
npm install -g serve
```

ולאחר מכן:

```
serve -s build
```

וכניסה אל:

```
Localhost:5000
```

ההבדל בין השיטה הזו להקלדת npm start הוא שכאן אנו מריצים את הבילד ממש, כלומר את הקבצים המוגמרים שאנו אמורים להעלות לשרת. מודול serve יוצר לנו שרת ומעלה את התיקייה build כדי שנוכל לבדוק את האפליקציה שלנו, מתיקית ה-build, מקומית.

שימו לב: אם כאשר אתם מקלידים serve אתם מקבלים הודעת שגיאה בנוסח:

```
serve : The term 'serve' is not recognized as the name of a cmdlet
```

עשו את הפעולות הבאות:

הקלידו:

```
npm install serve
```

(ללא ה g-)

ולאחר מכן הקלידו:

```
node .\node_modules\serve\bin\serve.js -s build
```

תרגיל:

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

פתרון:

יש להריץ npm run build ולחכות לתוצאה. אחרי שהתהליך מושלם, תיווצר אצלכם תיקיית build עם קובץ HTML, קובצי תמונה, CSS והכי חשוב: ג'אווהסקריפט.

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

offer@webits.com

פרק 25

סיום - ומה עכשיו?

סיום – ומה עכשיו?

אם קראתם את כל פרקי הספר, אתם יודעים לבנות אפליקציות ריאקטיות. אבל הידע שלכם הוא עדיין תיאורטי, ולימוד פריימוורק חדש נעשה באמצעות הידיים. אי-אפשר ללמוד פריימוורק אם לא עברתם את השלבים שאף ספר לא יכול להעביר אתכם: התסכול המתלווה למשהו שלא עובד כשורה, השמחה הגדולה כאשר מצליחים לפתור את התקלה ורוכשים ידע נוסף יקר מפז, החיפוש המתמיד בגוגל, בכלי בינה מלאכותית וב-stackoverflow ובניית מוצרים אמיתיים ופתרון התקלות שיש בהם.

אם סיימתם לקרוא את הספר ולפתור את התרגילים, הצעד הבא שלכם הוא לבנות אתר, אפליקציה או יישום מבוססי ריאקט. זה הזמן להצטרף לקבוצה שבונה אתרים או אפליקציות בהתנדבות עבור מיזמים הקרובים לליבכם, לבנות אתר או אפליקציה לחבר שתמיד היה לו רעיון או לעמותה שתמיד רציתם לתרום לה. הניסיון המעשי הראשון שלי בבניית אתרים ובכתיבת קוד היה עבור עמותת טולקין (מחבר "שר הטבעות") בישראל – אין סיבה שזה 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 תכנות חוסכים שעתיים של תכנון. חשבו על הקומפוננטות, בדקו באילו קומפוננטות חיצוניות אפשר להשתמש ואיך הן עובדות ואם אפשר להשתמש בהן עבור האתר שלכם. נוסף על כך, בקשו לקבל ביקורת על העבודה שלכם וכך תגלו איך הייתם יכולים לעשות אותה טוב יותר.

אבל אחרי שבניתם את הפרויקט שלכם – יש עוד כמה צעדים שתצטרכו לעשות כדי להשתלב בשוק העבודה. אף אחד לא יקבל אתכם לעבודה על סמך קריאת הספר הזה ופרויקט גמר, מוצלח ככל שיהיה. יש כמה נתיבים נוספים שיאפשרו לכם להמשיך ללמוד ולצבור ניסיון – עוד לפני העבודה הראשונה. בספר "פיתוח ווב מעשי בעברית" אני מסביר על כיצד יוצרים ניסיון כזה ומשלבים אותו בקורות החיים.

התחברות לקהילת הפיתוח

אחרי שבניתם את הפרויקט שלכם, או במהלך הפיתוח שלו – חשוב גם להתחבר לקהילת הפיתוח הישראלית. היא חמה ונעימה ויש לא מעט מתכנתים שיוכלו לסייע לכם להמשיך הלאה. בפייסבוק יש קבוצה פעילה מאוד של מתכנתי ריאקט שפועלת בעברית וכדאי להצטרף אליה ולהשתתף (בענווה הראויה) בדיונים, לקרוא, ללמוד ולהבין לאן כדאי להתקדם, ואם יש פרויקט פתוח שמחכה למתנדבים – נסו להצטרף אליו.

מפגשים ומיטאפים

באתר meetup.com יש לא מעט מידע על מפגשים בחינם שנקראים מיטאפים. במפגשים האלו נפגשים מתכנתים ושומעים הרצאות או בונים יחד דברים. יש מפגשים המיועדים למתכנתי ריאקט וכדאי להירשם לקבוצות השונות שמארגנות מפגשים על מנת לדעת על אלו שמתקיימים בסביבתכם ובשעות המתאימות לכם. במפגשים האלו אפשר גם להמשיך ללמוד על פיצ'רים שונים בריאקט וגם להיפגש עם מתכנתים מקצועיים, לשאול לעצתם ולקבל הכוונה בנוגע להתקדמות בלימוד השפה ובפרויקטים שונים. מדי פעם גם יש באתר הזמנה להאקתונים ולמפגשים שבהם מתכנתים בפועל. כדאי מאוד להצטרף לקבוצה כזו ולתכנת לצד מתכנתים אמיתיים בליווי הכוונה אמיתית. כמה שעות עבודה כאלו יקדמו אתכם באופן מטאורי, גם אם לא תזכו בפרס.

האתר Stackoverflow

האתר Stackoverflow 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 מתכנתים 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-Stackoverflow הוא משהו שראוי לציין בקורות החיים.

תרומת קוד בגיטהאב

לבסוף, אפשר לתרום קוד בגיטהאב לפרויקטים שמשתמשים בריאקט. למשל, ספריית קומפוננטות שמבוססת על ריאקט. בגיטהאב יש לא מעט פרויקטים שמשוועים לתרומת קוד. במהלך הפרויקט שבניתם בוודאי ראיתם ספריות קומפוננטות שחסרה להן בדיוק הקומפוננטה הזו שהייתם רוצים. למה שלא תוסיפו אותה? או תתקנו את הדוקומנטציה? או תכתבו בדיקה אוטומטית בַּמקום שבו היא חסרה – תרומת הקוד הזו היא ממש בבחינת ניסיון, ואם תבחרו בנתיב הזה ותצליחו בו, תוכלו אפילו להשתלב בקלות בתעשיית התוכנה.

מצד שני, העבודה בגיטהאב מצריכה ידע בגיט, הבנה באנגלית ויכולת לתקשר עם המתכנתים שעובדים בפרויקט ומגיעים מכל העולם. יש גם מיטאפים שבהם מתכנתים אחרים מסבירים איך עובדים עם גיטהאב ותורמים קוד. בספר שלי "ללמוד פיתוח מעשי בעברית" אני מלמד גם על גיט ואיך תורמים קוד לגיטהאב על מנת לבנות תיק עבודות מסודר.

כך או כך – הספר הזה הוא הצעד הראשון בכניסה לעולם המרתק של ריאקט ופיתוח צד לקוח. אני מאחל לכם הצלחה בלימוד וגם בהשתלבות בתחום.

נספח: PropType

מחבר: רן בר-זיק עבור חברת HoneyBook

התקשורת בין קומפוננטות שונות נעשות באמצעות props שמעבירים קלט או, באמצעות פונקציות, פלט. על מנת להשתמש בקומפוננטה, אנו צריכים לדעת באיזה props להשתמש. מה קורה אם מי שמשתמש (או, במילים מקצועיות יותר, צורך) את הקומפוננטה שלנו לא משתמש ב-props המתאימים? הקומפוננטה לא תעבוד או שתעבוד באופן לא צפוי. אם אנו כותבים קומפוננטה, החובה שלנו היא לסמן לאלו שמשתמשים בה (גם אם המשתמשים בה הם אנחנו, בעתיד) באיזה props אפשר להשתמש. כדי שמי שמשתמש בקומפוננטה יקבל חיווי מיידי. איך עושים זאת? באמצעות PropTypes. מודול שהיה חלק מריאקט עד גרסה 15.5 והחל מהגרסה הזו הוא יצא למודול משלו שניתן להתקינו בנפרד. PropTypes הוא הדרך שלנו להצהיר איזה props אנו נדרשים לקבל ומאיזה סוג. כך למשל, אני יכול להצהיר על props מסוימים שנדרשים ועל כאלו שהם רק אופציונליים וכמובן לפרט את הסוגים שלהם. כך, אם אם מי שמשתמש בקומפוננטה שוכח להעביר props מסוימים, הוא יקבל התראה. אם הוא מעביר מחרוזת טקסט בעוד אנו מצפים למספר, הוא יקבל גם כן התראה מאוד ברורה.

למרות ש-PropTypes אינם כבר חלק מריאקט, רבים מהמתכנתים שמפתחים בריאקט עובדים איתם ומומלץ מאוד לעשות כן.

התקנת PropTypes

בדיוק כמו ספרית קומפוננטות חיצונית, שלמדנו עליה בעבר, או Create-React-App, אנו מתקינים את PropTypes באמצעות Node.js והכלי שלו להתקנת מודולים: npm. בהנחה ש-Node.js מותקנת במחשבכם ואתם משתמשים באפליקצית Create-React-App, הכנסו באמצעות הטרמינל לתיקיה הראשית של האפליקציה שלכם והקלידו את הפקודה הבאה:

```
npm install prop-types
```

אם הכל תקין, לאחר כדקה תוכלו לראות בטרמינל חיווי על כך שההתקנה הצליחה. מהנקודה הזו תוכלו להשתמש ב- PropTypes.

```
➜ my-app git:(master) ✗ npm install prop-types
npm WARN @typescript-eslint/eslint-plugin@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN @typescript-eslint/parser@1.13.0 requires a peer of eslint@^5.0.0 but none is installed. You must install peer dependencies yourself.
npm WARN ts-pnp@1.1.2 requires a peer of typescript@* but none is installed. You must install peer dependencies yourself.
npm WARN tsutils@3.17.1 requires a peer of typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-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 מייבאים את PropTypes באמצעות import, בדיוק כמו כל ספריה חיצונית אחרת שלמדנו עליה ובאמצעות האובייקט שאנו מייבאים אנו מצהירים על ה-props השונים.

הבה ונדגים באמצעות הקומפוננטה הפשוטה הזו:

```
function Welcome(props: {name:string}):JSX.Element {

  return (
    <p>Hello, {props.name} !</p>
  );
}
export default Welcome;
```

אם אנו רוצים להשתמש בקומפוננטה הזו, אנו נעשה זאת, למשל, כך:

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

function App(): JSX.Element {
  const userName:string = 'Moshe';
  return (
    <div className="App">
    <header className="App-header">
      <Welcome name={userName} />
    </header>
    </div>
  );
}

export default App;
```

אבל מה קורה אם אנו לא מעבירים את ה-props המתאימים? למשל אם אני לא מעביר name? זו תקלה שיכולה להתקיים. מהשלב שבו כתבתי את קומפוננטת Welcome עד השלב שאני משתמש בה יכולים לעבור חודשים, אולי שנים. אם אשכח את מה שאני אמור להעביר - מה שיקרה הוא שהקומפוננטה לא תעבוד כשורה ולי לא יהיה מושג למה. אני אצטרך לנבור בקוד כדי להבין מה השתבש.

הבה ונשתמש ב- PropType. ראשית ייבוא ואז הצהרה:

```
import PropTypes from 'prop-types';


function Welcome(props:{name: string}):JSX.Element {

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

Welcome.propTypes = {
  name: PropTypes.string.isRequired,
}

export default Welcome;
```

הייבוא נראה כך:

```
import PropTypes from 'prop-types';
```

ההצהרה נראית כך:

```
Welcome.propTypes = {
  name: PropTypes.string.isRequired,
}
```

ההצהרה 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 מדובר ב-prop שהוא נדרש לפעולה תקינה של הקומפוננטה.

אם אני אנסה להשתמש בקומפוננטה בלי להצהיר על name. אני אקבל הודעת שגיאה מאוד ברורה שמודיעה לי על כך ש-prop name מסומן כנדרש אך הערך שלו הוא לא מוגדר.

```
▶Warning: Failed prop type: The prop `name` is marked as required in `Welcome`, but its value is `undefined`.
    in Welcome (at App.js:10)
    in App (at src/index.js:16)
    in Provider (at src/index.js:16)
```

אם אני אעביר סוג נתון שאינו תואם לסוג שאני מבקש, אני אקבל גם הודעת שגיאה ברורה שמודיעה לי ש-prop name קיבל ערך שאינו תואם את מה שהקומפוננטה רוצה לקבל.

```
▶Warning: Failed prop type: Invalid prop `name` of type `number` supplied to `Welcome`, expected `string`.
    in Welcome (at App.js:10)
    in App (at src/index.js:16)
    in Provider (at src/index.js:16)
```

למרות הודעת השגיאה – הקומפוננטה תמשיך לרוץ כרגיל ותנסה לרנדר את הפלט. אם לא יהיו שגיאות נוספות – השגיאות ש- PropTypesמציגה לא יגרמו לקומפוננטה לא לרוץ.

ערכים שאפשר לקבוע

אנו יכולים להודיע באמצעות PropTypes על מגוון רב של סוגי נתונים שונים – כמעט כל הנתונים הפרימיטיביים האפשריים. בטבלה הזו יש ריכוז של הסוגים הנפוצים:

סוג הערך	תיאור	דוגמה
מחרוזת טקסט	מחרוזת טקסט פשוטה	PropTypes.string
מספר	כל מספר שהוא	PropTypes.number
בוליאני	falseאוtrue	PropTypes.bool
פונקציה	כל פונקציה, רגילה או אנונימית	PropTypes.func
מערך	מערכים המכילים מידע, כולל מערך ריק	PropTypes.array
אובייקט	אובייקטים שמכילים מידע או מתודות, כולל אוביקטים ריקים	ropTypes.object
אלמנט ריאקט	קומפוננטה ריאקטית כלשהי שמועברת כפרמטר	PropTypes.element

אני יכול להחליט שה-PropType שלי יקבל כמה סוגים. כך למשל, אם הקומפוננטה שלי יודעת להתמודד עם מספר או טקסט ב-prop מסוים. אני יכול לציין שאני יכול או סוג אחד, או סוג שני. אני עושה את זה באמצעות הפקודה oneOfType במקום הסוג.

למשל:

```
import PropTypes from 'prop-types';


function Welcome(props: {name: string}): JSX.Element{

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

Welcome.propTypes = {
  name: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number,
  ]).isRequired,
}

export default Welcome;
```

אני תמיד יכול לשרשר את isRequired במידה והפרמטר נדרש. אך זו לא חובה.

חשוב להשתמש ב-PropTypes. מדובר בתוספת קטנה וחשובה מאוד לכל קומפוננטה שיכולה לעשות סדר בקוד. נכון, זה מעט יותר עבודה, אבל לטווח ארוך זה חשוב וכדאי.

נספח: שינויים מהמהדורה הקודמת (מהדורה 1.1.1)

בגרסה הוכנס שינוי משמעותי: מלמדים ריאקט עם טייפסקריפט ולא עם ג'אווהסקריפט. כל הדוגמאות הוחלפו.
הוכנסו שדרוגים והתאמה לריאקט 18 – החלפת מתודות ישנות, הסבר על כך שעדיף שלא להשתמש בקלאסים.
הוכנסו שדרוגים המתאימים לגרסה האחרונה של react-router ו-material ui (מאי 2023).

הוסר הפרק על רידקס כיוון שהפופולריות של רידקס כבר לא בשיאה, הוא מורכב מדי ללומדים חדשים וקונטקסט כבר פופולרי בהרבה. באופן אישי למדתי לתעב את רידקס בלהט, אני כבר לא יכול ללמד אותה.

נספח: שינויים בין מהדורה 1.1.1 למהדורה 1.0.0

פרק הרידקס הוחלף לחלוטין בדוגמה פשוטה יותר. בדוגמה החדשה אין אובייקט list שבתוכו מערך של items אלא מערך של items בלבד. בנוסף, הוכנה אפליקצית ריאקט ב sandbox.io וקישור אליה הוצב בסוף הפרק.

נוסף תרגיל לפרק הרידקס.

הדוגמה בפרק על הסטייט הוחלפה לדוגמה הכוללת אירוע ולחיצה במקום דוגמת setInterval.

תיקוני שגיאות הקלדה בפרק אודות המרצים ואודות Honeybook.

עדכון הדוגמאות והקישורים בפרק הפתיחה לריאקט גרסה 17.

הורדת import React מהדוגמאות בשל השדרוג לגרסה 17.

הוספת משפט ודוגמה ב-React.Fragment על הקיצור המקובל: <>.

תודה רבה על הרכישה של הספר!

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

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

העותק הזה נמכר ל:

omer@webiks.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נחבאים פרטי הרוכש באופן שקוף למשתמש. כדאי מאוד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למישהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיחקו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!