Contents

הקדמה לReact

מה זה React?

בכמה מילים: ריאקט היא ספרית קוד פתוח לפיתוח אפליקציות web (אתרים) בצד ה-client. ריאקט היא ספרית components (קומפוננטות) – רכיבים ויזואליים שחוזרים על עצמם באפליקציה שלנו, במספר מקומות.

אפשר להגדיר קומפוננטה כתגית HTML חדשה.

הרבה משווים את ריאקט לאנגולר. יש בינהן דמיון אולם הן שונות מאד. ריאקט, בניגוד לאנגולר, היא סה"כ ספרית VIEW ולא כוללת בתוכה את שאר האפשרויות, כמו אנגולר (MV)

יוצר ריאקט, הינו אדם בשם Jordan Walke, מהנדס תוכנה בפייסבוק שרצה לפתח בצורה קלטה ופשוטה יותר מערכות ואפליקציות ווב. והכי חשוב, שמהירות האפליקציה תהיה מהירה פי כמה מספריות אחרות בתחום. ספרית ריאקט כיום נמצאת בשימוש במגוון רחב של אתרים.

התחזוקה, הפיתוח והקידום של ריאקט כיום הוא ע"י פייסבוק עצמם.

למה ריאקט טוב יותר?

קשה להחליט חד משמעית שריאקט טוב יותר מפריימוורק (FrameWork) ענק ורוחבי כמו אנגולר, זאת טכנולוגיה שעובדת שונה לחלוטין, אבל ריאקט מהיר יותר, ומשמעותית קל יותר להבנה.

אחד השינויים בין ריאקט לאנגולר הוא unidirectional data flow. בניגוד לאנגולר, כל הדטה מועבר מלמעלה למטה – קומפוננטת האב מעבירה מידע לבן, הבן לנכד וכן הלאה.

זה נשמע קצת יותר מסובך מאנגולר, שנותן בקלות לערוך את הstate של כל אלמנט, ומכל קומפוננטה, אבל זה יותר בטוח. אתם לעולם לא תשנו אלמנט, אלא רק במידה ואתם באמת רוצים לשנות אותו. ה-unidirectional data flow של ריאקט מוסיף מהירות גדולה, יותר שליטה ויותר הבנה במה שהקוד שלכם עושה.

בנוסף, שינוי משמעותי ריאקט בניגוד לאנגולר שעובד בצורה דומה לjquery, אשר משנה את הDOM הקיים, כלומר הוא טוען בהתחלה את כל הHTML, ולאחר מכן מבצע בו שיניים. ריאקט עובד על virtual dom – שנותן לו את הכוח והמהירות הגדולה

מה זה virtual dom?

הדום הוירטואלי של ריאקט, הדבר הראשון החשוב ביותר בדום של ריאקט הוא שכל קוד הHTML, נכתב מתוך הjavascript. אין יותר הפרדה בין (html,js(css, הכל משתלב לקובץ JS אחד (או במקרה של ריאקט – להמון קבצי JSX/JS) המכיל גם את הHTML, וגם את הJS

מאז ומתמיד שאנחנו משתמשים בJS, אנחנו מבצעים מניפולציה כלשהי בDOM הקיים, שזאת בעצם עריכה של הHTML. תמיד הייתה הפרדה בקבצים, אבל השילוב נהיה מסובך יותר, ואיטי יותר למשתמש, כאשר אנחנו רוצי ליצור כמה שיותר מניפולציות, כמו בכל אפליקציית ווב.

שערכנו את הDOM הקיים דרך הJS, תמיד היינו צריכים למצוא אלמנט מסוים בדף, לבצע עליו מניפולציה כלשהי ולהחזיר את התשובה החדשה למשתמש. מובן בהחלט אם כך, שכל התהליך המסורבל הזה יתבצע דרך קובץ אחד הנותן שליטה מלאה גם על המניפולציה, עם על הפעולה וגם על מה שיוצג למשתמש.

ה- Virtual DOM הוא מבנה שמסודר גם הוא בעץ, כמו HTML, אבל בניגוד ל HTML ל Virtual DOM של ריאקט יש יכולת שנותנת לו את הכוח המרכזי - היכולת לזהות במהירות את ההבדל בין שני Virtual DOMs.

בהינתן שני עצי Virtual DOM שיכולים לכלול הרבה מאוד אלמנטים, ריאקט מצליחה בצורה מאוד מהירה לזהות מה ההבדל בין האחד לשני, ואיזה פקודות JavaScript צריך לכתוב כדי לעבור בין המבנה הישן למבנה החדש, לכן שיטת העבודה של ריאקט עם פקדים היא:

1. ריאקט מפעיל את פונקציית הפקד (הפונקציה render אם זה קלאס, או פשוט הפונקציה שהיא הפקד) ומקבל ממנה מבנה שנקרא Virtual DOM.
2. ריאקט "זוכרת" איזה Virtual DOM נמצא כרגע על המסך, ומחשבת מהר את הדרך הקצרה ביותר להגיע ממה שמוצג עכשיו על המסך לתוצאה שהיא חישבה בסעיף (1). לאחר מכן ריאקט מייצרת פקודות JavaScript שיעברו למצב החדש.
3. בכל פעם שמשתנה State של אחד הפקדים, באופן אוטומטי ריאקט מפעילה מחדש את פונקציית הפקד כדי לקבל את ה Virtual DOM החדש אחרי עדכון ה State. לאחר מכן כמו בסעיף (2) ריאקט מזהה את ההבדלים בין מה שרואים עכשיו על המסך לבין ה Virtual DOM החדש שקיבל ומייצרת את ההוראות כדי לעבור למצב החדש וחוזר חלילה.

הVirtual DOM , מעבר למה שהוא נותן לנו אפשרות של עריכת ה-HTML בתוך הJS עצמו, הוא גם עושה את זה בצורה חכמה ומהירה הרבה יותר. אך כמובן, ליצור כל פעם פונקציה על מנת ליצור אלמנט HTMLחדש זה לא פשוט, לכן בריאקט יצרו את JSX – שפת כתיבת הHTML בתוך הJS

מה זה JSX?

JSX היא טכנולוגיה חדשה שנוצרה ע"י פייסבוק, לכתיבה פשוטה של HTML בתוך JS, וספציפית יותר בתוך קוד של ריאקט

במקום להשתמש בפונקציות React.createElement(element, props, children) ליצירת אלמנט HTML לדוגמה:

```
React.createElement("ul", null,
    React.createElement("li", null, 'option 1'),
    React.createElement("li", null, 'option 2'),
    React.createElement("li", null, 'option 3'),
    React.createElement("li", null, 'option 4'),
    React.createElement("li", null, 'option 5')
)
```

שכאן אנחנו בעצם יוצרים ul ובתוכו 5 li עם התוכן option X.

אנחנו נוכל לכתוב את הקוד הנ"ל, בJSX בצורה הבאה:

```
<ul>
```

```
<li>option 1</li>
<li>option 2</li>
<li>option 3</li>
<li>option 4</li>
<li>option 5</li>
```

וזהו קוד שנמצא בתוך הJS, ונכתב באמצעות JSX, ממש HTML.

מכיון שהדפדפנים לא יודעים לקרוא JSX, לפני הרצת הקוד נהיה חייבים לקמפל את הקוד הזה לJS תקין באמצעות מס' טכנולוגיות אפשריות (בקורס נשתמש בNPM

הקומפוננטות בריאקט

בריאקט אנחנו עובדים בצורה של Separation of concerns. כל אזור באפליקציה , כל קומפוננטה נוצרה על מנת לדאוג לדבר אחד בלבד. מעבר לזה, אנחנו יכולים ליצור קומפוננטה אחת בלבד, וליישם אותה במגוון רחב של אפליקציות. מאחורי ריאקט בנויה מאד בצורה של OOP.

כל אפליקצית ריאקט תהיה מחולקת לכמה שיותר קומפוננטות נפרדות, שלכל אחת מהן תהיה פעולה משלה. אלו תהינה תחומות בתוך קומפוננטת האב, האב בתוך הסב וכן הלאה, עד שמגיעים למוקר ליבה של האפליקציה – הקומפוננטה הראשית, שהתחילה את הכל. לרוב תקרא Index או App.

מה היתרונות של זה?

1. אם יש תקלה בקומפוננטה כל שהיא, קל מאוד להגיע לתקלה ולתקן את הבעיה. אין צורך לעבור על אלפי שורות של קוד, על מנת להגיע לתקלה. מספיק לעבור על אותה קומפוננטה אשר אחראית לתקלה הספיציפית שנוצרה. ככה ניתן לעבור על קובץ של קומפוננטה שאולי כולל סה"כ 100 שורות של ולפתור בזריזות את התקלה.

2. קל להעביר קומפוננטות מסויימות שיוצרים, למערכות אחרות שמפתחים – או אפילו לשתף את הקומפוננטות עם מפתחים אחרים של React. כמו שאתם תוכלו לשתמש בקומפוננטות שאחרים יצרו, כך גם הם יוכלו להשתמש בקומפוננטות שלכם.

הכנת סביבת הפיתוח בריאקט

כדי לכתוב ולהריץ על המחשב אפליקציית ריאקט יש לוודא התקנות של הרכיבים הבאים:

1. Node.js – ניתן להוריד מהאתר הרשמי – קישור ישיר לדף ההורדה – יש לבחור את הגרסה המתאימה לפי מערכת ההפעלה (מומלץ להוריד את קובץ ה msi). ההתקנה פשוטה, לחיצה על next עד לסיום. (התקנה זו תתקין גם את מערכת ה NPM (node package manager) אותו נצטרך כדי להתקין את חבילת יצירת פרויקט ריאקט, וכדי להריץ את הפרויקט)
2. התקנת "יצירת פרויקט ריאקט" – לאחר התקנת Node.js (ניתן לוודא התקנה ע"י שימוש בפקודה node –version דרך ה-CMD – שורת הפקודה. במידה שיש פלט של מס' גרסה – הNODE מותקן) יש להריץ בCMD את הפקודה *npm install npx create-react-app*.
3. התקנת VSCODE (visual studio code) – ניתן להוריד מהאתר הרשמי שלו – קישור ישיר לדף ההורדה - יש לבחור את הגרסה המתאימה למערכת ההפעלה. גם התקנה זו פשוטה, ללחוץ על next עד לסיום. לפני סיום ההתקנה, ישנן מס' העדפות שכדאי לשים לב אליהן.

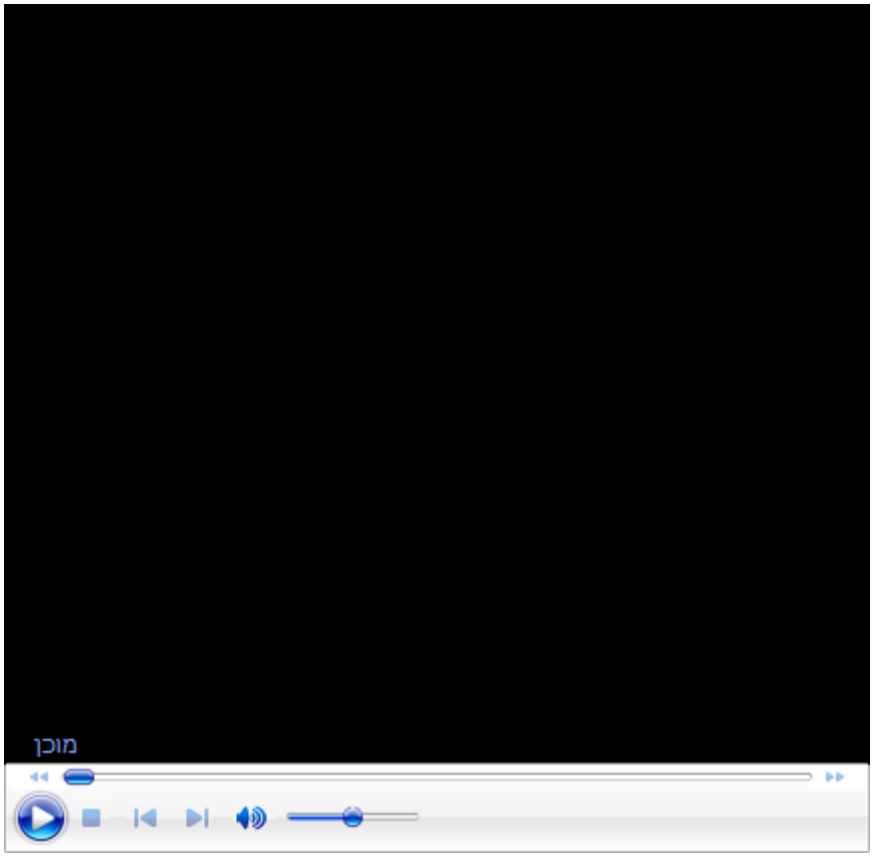
אחת המרכזיות היא האפשרות לפתוח קובץ/תקיה באמצעות תפריט של מקש ימני על התקיה/הקובץ.

אם מותקנים כל הרכיבים הנ"ל, ניתן להתחיל לכתוב בריאקט...

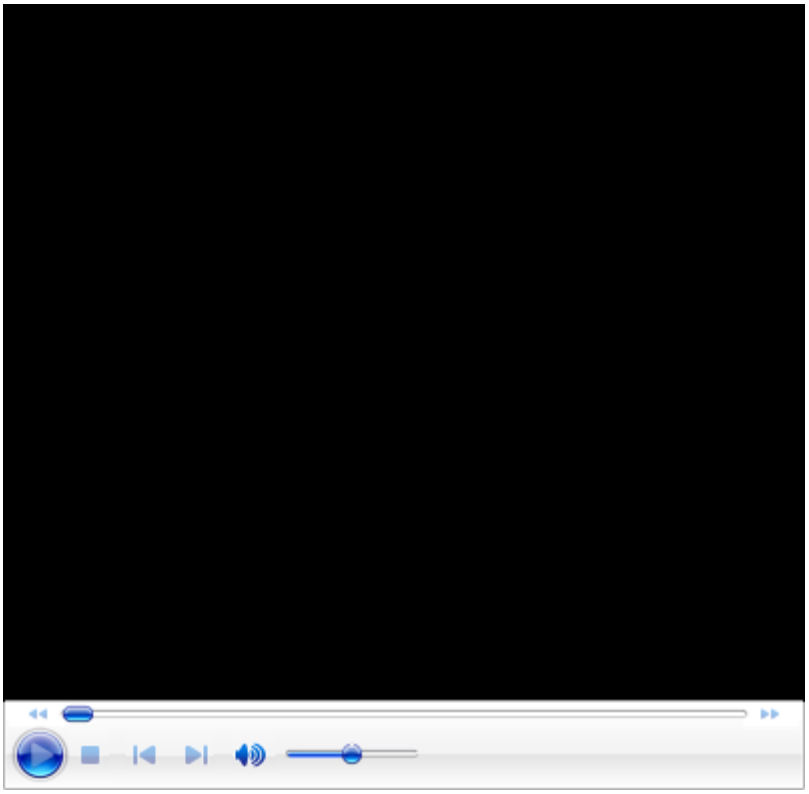
אז איך מתחילים:

1. יצירת פרויקט חדש: כדי ליצור פרויקט בריאקט, נשתמש בהרצת פקודה דרך הCMD, בניתוב בו נרצה ליצור את הפרויקט.
איך פותחים CMD בניתוב מסוים:

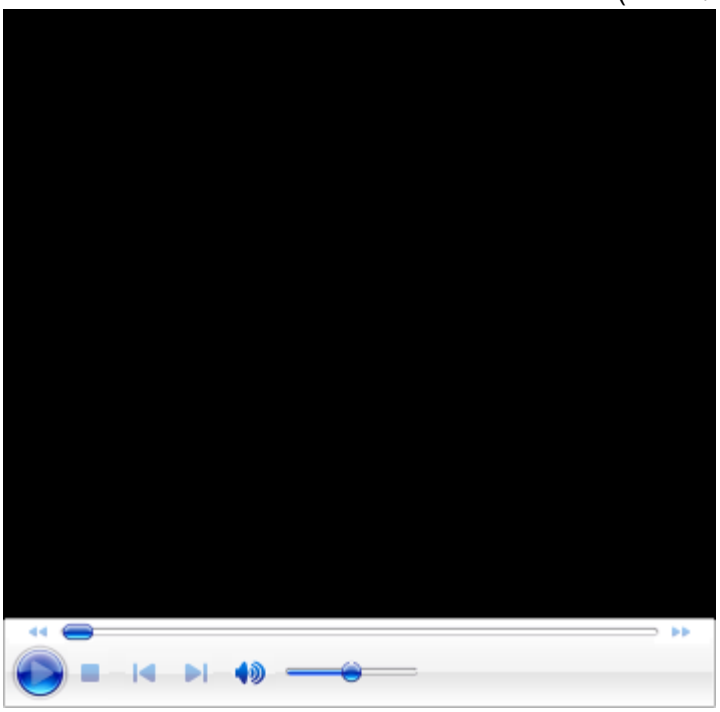
- כניסה לתקיה המבוקשת דרך סייר הקבצים, לחיצה על ניתוב התקיה, יש למחוק את הניתוב ובמקומו לכתוב CMD -<ENTER. חלון הCMD יפתח בניתוב המתאים.



- העתקת הניתוב (באותה צורה – לחיצה כפולה על ניתוב התקיה) פתיחת ה cmd-> cd -> הדבק



- ניתוב יחסי באמצעות פקודת cd (העלאת רמה באמצעות /.., השלמה אוטומטית באמצעות Tab)



כאשר הCMD מנותב לתקיה הרצויה (בכל אחת מהדרכים) ניצור את הפרויקט באמצעות הפקודה

npx create-react-app <project>

(את ה<project> - כולל הסוגריים הזויתיות, יחליף שם הפרויקט שלכם)

שימו לב! שם הפרויקט אינו יכול להכיל אותיות גדולות.

נ.ב. שגיאה נפוצה במהלך יצירת הפרויקט.

אם במהלך יצירת הפרויקט מופיעה השגיאה הבאה:

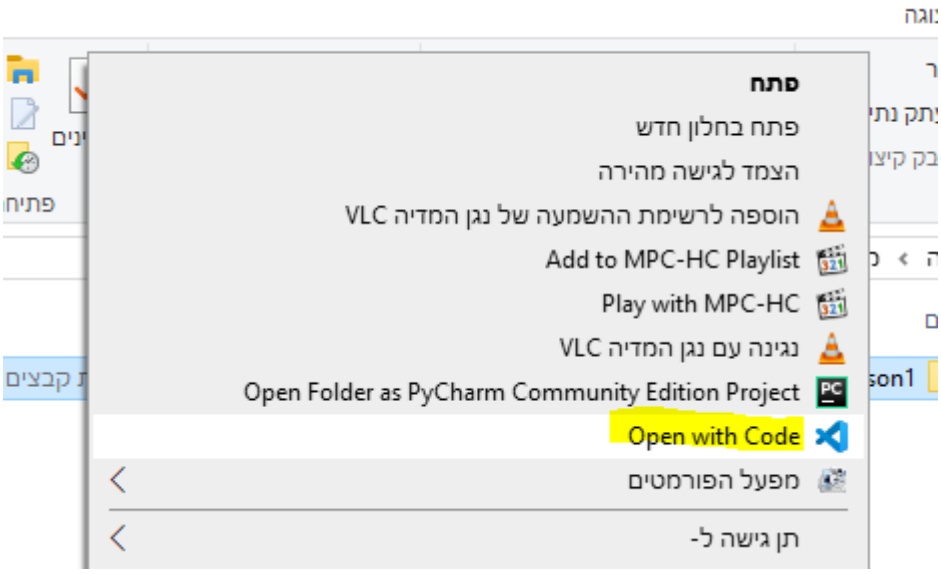
```
Select npm

Creating a new React app in C:\'[illegible] [illegible] [illegible]\react\myproject.

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

npm ERR! Unexpected end of JSON input while parsing near '                    '

npm ERR! A complete log of this run can be found in:
npm ERR!     C:\Users\DELL\AppData\Roaming\npm-cache\_logs\2020-09-15T18_07_05_724Z-debug.log

Aborting installation.
  npm install --save --save-exact --loglevel error react react-dom react-scripts cra-template has failed.

Deleting generated file... package.json
Deleting myproject/ from C:\'[illegible] [illegible] [illegible]\react
```

ניתן לפתור את זה בצורה הבאה – יש לגשת דרך סייר הקבצים ל:

C:\Users\<MyUser>\AppData\Roaming ולמחוק את התקיות npm,npm-cache. אם התקיה AppData אינה מופיעה, יש להציג קבצים מוסתרים במחשב.

איך פותחים את הפרויקט?

- אם במהלך התקנת הVSCODE אפשרתם פתיחת תקיה דרך תפריט ימני – ניתן בדרך זו.



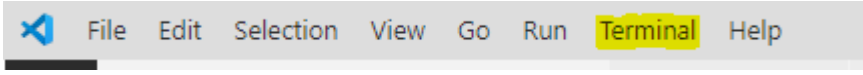
- ניתן לפתוח ע"י הCMD – ניתוב **לתוך** תקית הפרויקט, באחד מהדרכים שצוינו לעיל, בתוך הCMD יש לכתוב את הפקודה . code (כולל הרווח והנקודה) -> ENTER.

הפרויקט יפתח בVSCODE.

איך מריצים את הפרויקט.

לאחר שפתחנו את הפרויקט בVSCODE, יש לפתוח בו את הTERMINAL ומשם להריץ את הפקודה.

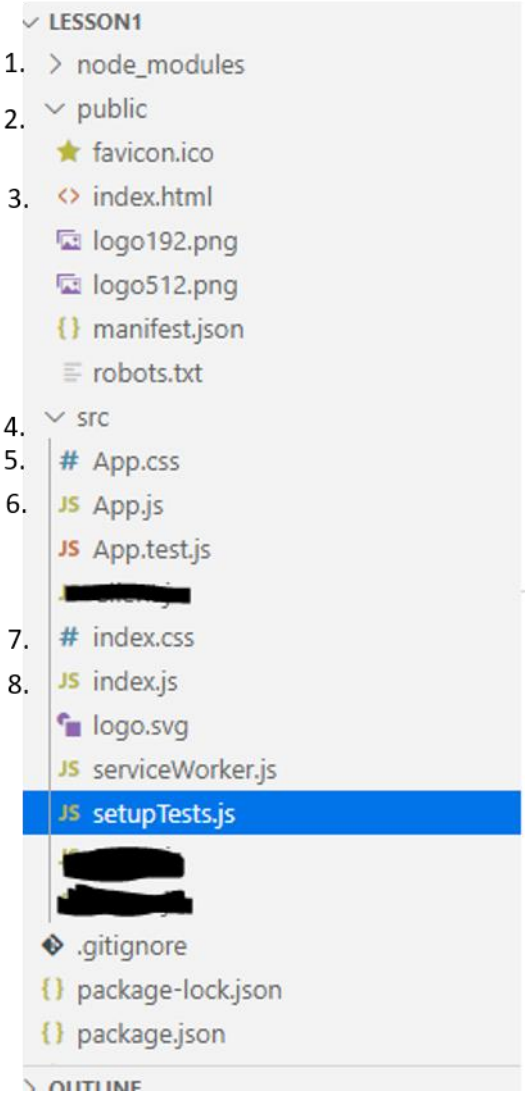
ENTER <- npm start



הפרויקט יעבור קומפליציה, ירוץ ויפתח בדפדפן (ברירת המחדל) בכתובת http://localhost:3000

מבנה פרויקט בריאקט:

1. תקית node_modules מכילה את כל המודולים הנדרשים לריצת הפרויקט
2. תקית הpublic- תכיל משאבים כללים לפרויקט (כמו תמונות וכו'), בתקיה זו נמצא גם:
3. קובץ הindex.html שהוא הדף הראשי של הפרויקט
4. תקית src בה יהיו קבצי המקור של הפרויקט אותם נערוך/נוסיף חדשים
5. App.css קובץ CSS עבור הקומפוננטה App
6. App.js - קובץ הקומפוננטה App
7. Index.css – קובץ CSS עבור האפליקציה כולה
8. Index.js קובץ ה"boot" ממנו מתחילה לרוץ האפליקציה.



בואו נתחיל לכתוב אפליקציה בריאקט:

כאשר יצרנו פרויקט, הוא נוצר בברירת המחדל עם הקומפוננטה App

בתחילה הקומפוננטה מכילה את הקוד הבא:

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

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

  </div>
 );
}

8. export default App;
```

הסבר שורות מיוחדות בקובץ

1. ייבוא ה'רכיב' React מספרית react
2. ייבוא קובץ הSVG המוצג על המסך
3. ייבוא קובץ הCSS המשותף לקומפוננטה זו בלבד
4. הצהרה על הקומפוננטה בשם App
5. החזרת התצוגה של הקומפוננטה
6. שימוש בclassName במקום class בhtml הרגיל
7. שימוש בתחביר הצומדיים כדי "לשתול" ערכי JS בתוך ה HTML שלנו.
8. "ייצוא" הקומפוננטה שלי, מאפשר לנו להשתמש בה במקום אחר.

מה זה בדיוק קומפוננטה בריאקט:

קומפוננטה מייצגת לי תצוגה ויזואלית והתנהגות של פקד.
מסיבה זו, קומפוננטה תכיל לי פונקציות לניהול ה"התנהגות" של הפקד, ותחזיר רכיב/י HTML שמתארים את תצוגת הקומפוננטה.

כדי לאפשר לקובץ להכיל קומפוננטה, יש לייבא בו את הרכיב React מהמודל React – בסעיף 1.
קומפוננטה, בצורה המקובלת כיום, מיוצגת באמצעות פונקציה פשוטה. כדי "לייחצן" את הפונקציה – קומפוננטה שלי – נוסיף את המילים export default לפני הצהרת הפונקציה, או לאחר סיום הפונקציה כמו בסעיף 8. (האפשרות הראשונה היא export default funcation App)
קומפוננטה מחזירה לי רכיבי HTML המתארים את המראה הויזואלי של הקומפוננטה.

הערה חשובה!

ערך מוחזר מקומפוננטה, חייב להיות במבנה XML, אשר מכיל תגית root אחת.
כלומר לא יתכן להחזיר מקומפוננטה את הרכיבים הבאים, לדוגמא:

```
<div>
</div>
<div>
<div>
```

כדי לפתור בעיה זו ישנן שתי דרכים אפשריות:

1. לעטוף את כל הרכיבים באלמנט "אבא" אחד (בד"כ div)
2. עטיפת הרכיבים ב fragment ריק – תגית html ריקה. – בד"כ נשתמש באופציה זו אם אלמנט מסוים יכול "להרוס" לנו את התצוגה. בדום שלנו, בדפדפן, הוא יתעלם מתגית זו, ויציג רק את הילדים שלה. לדוגמא:

```
<>
 <div>
</div>
 <div>
 <div>
</>
```

ישנם שני הבדלים (כרגע) בין html רגיל, לבין הhtml שנכתב בJSX

- כמו שראינו בסעיף 6, התחביר להוספת מחלקות CSS לאלמנט הוא className ולא class
- שימוש בתגית style יהיה במבנה הבא: style={{attr:value}} כמו כן, בשונה מCSS רגיל, בו מאפייני CSS בעלי שתי מילים יופרדו עם מקף – בריאקט (במידה שכותבים זאת בHTML ולא בקובץ הCSS!) יופרדו מאפיינים אלו עם אות גדולה בתחילת כל מילה, החל מהמילה השניה, לדוגמא: backgroundColor בריאקט, לעומת background-color בCSS רגיל.

פונקצית הקומפוננטה בריאקט עשויה להכיל בתוכה מס' רב של פונקציות לטפל בהתנהגות ובלוגיקה של הקומפוננטה.

מכיון שאחד הרכיבים המרכזיים בריאקט הוא מנגנון הtwo way binding – תלות דו כיוונית של ערכים בין תצוגת הUI לבין קוד הJS, יש צורך באפשרות "לשתול" ביטויי JS בתוך הJSX שלי (רכיבי הHTML)
אפשרות זו מתבצעת ע"י תחביר הצומדיים, כמו בסעיף 7.
כאשר רוצים "לשבור" את רכיבי הHTML ולשתול בו כל ביטוי JS תקין, מכניסים את הביטוי בתוך צומדיים (סוגריים מסולסלות, {}) , כל מה שיהיה כתוב בינהם, יתבצע כביטוי JS לכל דבר, בין אם מדובר במשתנה, תוצאה חישובית, ערך מוחזר מפונקציה ועוד.
צורה זו עשויה לבוא במעין שתי צורות:

1. Inner html של רכיבי html לדוגמא הכנסה של שם הנמצא במשתנה בתוך הJS בתוך תגית div יראה בצורה הזו: {name}

2. שליחה ערך לattribute של רכיב html-י . לדוגמא (מלבד הדוגמא בסעיף 7)   במקרה כזה, בשונה מattributes רגילים של html, נשמיט את המרכאות, ונשלח את הערך רק עם צומדיים. אם נשלח לתגית style אובייקט המתאר את "עיצוב" האלמנט, נשלח זאת רק עם זוג צומדיים אחד, ולא שתיים, כי אנחנו שולחים ערך, ולא מאפיין מאפיין. נקודה חשובה שיש לשים לב אליה, במידה ששולחים style בצורה כזו, היא שכל מאפייני האובייקט חייבים להיות זהים לשל ה style הרגיל של **ריאקט**.
לדוגמא:

```
  var style={
color:"red",
backgroundColor:"blue",
fontSize:"40px"
   }
<div style={style}></div>
```

איך יוצרים קומפוננטה חדשה בריאקט.

יצירת קומפוננטה בריאקט היא ע"י הוספת קובץ חדש בתקית הsrc, עם סיומת JS.
מקובל, ומאד כדאי, לתת שם שמתחיל באות גדולה, כמו כן, כדאי ששם הקומפוננטה יהיה כמו שם הקובץ.
קובץ הבסיס של קומפוננטה יכיל את השורות הבאות

```
import React from 'react';

export default function MyComponenet(){
    return <div>MyComponenet work</div>
}
```

איך נטמיע את הקומפוננטה שלי במקום אחר?

כאשר אנחנו יוצרים קומפוננטה, כמובן שהמטרה שלנו, שנוכל להטמיע אותה באפליקצייה שלנו. לכן, בקומפוננטה **שתכיל** את הקומפוננטה החדשה שיצרתי, יש צורך "לייבא" אותה ע"י שימוש בשורת הקוד 'import MyComponent from './MyComponent בראש הקובץ.

לאחר מכן, פשוט נטמיע אותה כמו כל אלמנט html אחר.
לדוגמא, קומפוננטת ה- App שלנו, תוכל להראות כך

```
import React from 'react';
import MyComponent from './MyComponenet'

export defualt function App() {
  return (
    <div className="App">
      <MyComponent/>
</div>
  );
}
```

שכפול אלמנטים בריאקט בלולאה.

לעיתים נרצה לשכפל אלמנט מסוים/קומפוננטה מסוימת באמצעות ריצה בלולאה, בד"כ על מערך (שימוש לדוגמא: כאשר יגיעו לנו נתונים מהserver – בהמשך, נרצה להציג אותם בצורה דינמית, וכו')

כדי לבצע זאת, נוכל להשתמש בפונקציה map על מערכים, בצורה הבאה, כמו הדוגמא של קומפוננטת הריבועים הצבעוניים שיצרנו

```
import React from 'react';
 export default function Squre(){
    var colors=["red","green","yellow","pink","black","gray","blue"];
    return colors.map(x=>(
    <div>
    </div>
    ))
}
```

תחביר השימוש בפונקציה :

Array.map(x=>(myHtmlElement))

כאשר x מהווה מעין item והפונקציה מהווה מעין לולאת foreach, אשר עוברת על כל אחד מאיברי המערך, ומחזירה עבורם אלמנט html. ניתן להשתמש ב x בכל מקום בתוך הלולאה, כפי שנראה בהמשך, לדוגמא, לשים כל inner html את שם הצבע מהמערך, בתוך תגיות הdiv, נכתוב זאת כך:

```
return colors.map(x=>(
    <div>
      {x}
    </div>
    ))
```

או נרצה לבצע השמה לצבע הרקע של הdiv, לצבע הנוכחי, נבצע זאת בצורה הבאה

```
return colors.map(x=>(
    <div style={{backgroundColor:x}}>
        {x}
    </div>
    ))
```

גם כאן, הx הוא ביטוי (משתנה) וניתן לשתול אותו כvalue של צבע הרקע והוא מופיע תחת צומדיים.

אם הלולאה מתחילה בתוך תגיות html, כיון שמדובר בקוד JS, נצטרך לשבור אותו באמצעות צומדיים. כמו כן, ניתן לקנן לולאות אחת בתוך השניה, לדוגמא בטבלה, בה נרצה לעבור על רשימת שורות ועמודות:

```
return (
        <>
           <table>
               {
                  mat.map(row =>
                      <tr >
                         { row.map(cell => <td>{{cell}} </td>) }
                      </tr>
                          )
               }
            </table>
        </>
    )
```

שימוש במטפלי אירועים:

כאשר נרצה ליצור מטפלי אירועים (event), נבצע זאת כמו בhtml עם שינוי קטן:
בhtml רגיל, כאשר רצינו לבצע פעולה מסוימת בעת אירוע, בתוך מטפל האירוע, קראנו לפונקציה אותה רצינו לבצע, לדוגמא: *<button onclick="buttonClicked()"> click me </button>*
מטפלי האירוע בריאקט, בשונה מהJS הרגיל, מצפים לקבל **פונקציה** ולא פקודות JS
לכן, במטפלי האירועים בריאקט, נשלח בד"כ את **שם הפונקציה** בתוך צומדיים לדוגמא:

<button onClick={buttonClicked}> click me </button>

גם כאן, מילה חדשה בשם מטפל האירוע, החל מהמילה השניה, בשונה מhtml רגיל – תתחיל באות גדולה, לדוגמא onClick בריאקט לעומת onclick בhtml רגיל.

מצב זה, של שליחת שם הפונקציה, מתאפשר רק במידה שהפונקציה שלנו לא מקבלת פרמטרים,

כלומר הצהרת הפונקציה נראית כך: function buttonClicked() {...}.

במידה והפונקציה שלנו מקבלת פרמטרים (הסתייגות: פרמטרים שאנחנו חייבים לשלוח – הסבר בהמשך) יש צורך לשלוח אותם בצורה שונה.

איך ניתן לשלוח? הרי מטפלי האירועים בריאקט מצפים לקבל פונקציה.

הפתרון: כתיבת פונקציה אנונימית, ובתוכה לשלוח לפונקציה הרצויה, עם רשימת הערכים.

מהי פונקציה אנונימית? פונקציה אנונימית היא פונקציה ללא שם, שניתן לקרוא לה רק מהמצביע שלה/ מההצהרה שלה, כמו במקרה הזה.

מהו תחביר פונקציה אנונימית?

ניתן לכתוב זאת בשתים (שלוש) צורות:

1. function () {...}
2. ()=>{...} – במקרה זה ניתן לוותר על על המילה function ובין הסוגריים של הפרמטרים לצומדיים של הפונקציה – נוסיף חץ – (=>). אם הפונקציה האנונימית שלנו מכילה רק פקודה אחת, לדוגמא קריאה לפונקציה ניתן לוותר על הצומדיים של הפונקציה ולכתוב כך :
3. ()=> jsCommand;

איך מבצעים את זה בפועל:

בתוך גוף הפונקציה האנונימית שלנו, נשלח קריאה לפונקציה שאותה נרצה לבצע:

לדוגמא: *<button onClick={()=>buttonClicked(1,2)}>*

כהמשך להסתייגות: תתכן פונקציה המצפה לקבל פרמטרים, ואנחנו **נרצה שהם יתקבלו** (כלומר פרמטרים בפונקציות בJS, הם אופציונליים, מבחינת התחביר. במידה שלא ישלחו מספיק פרמטרים – פרמטרים אלו בפונקציה יהיו undefined. אם ישלחו פרמטרים מיותרים, הם פשוט ימחקו) לעיתים פרמטרים אילו נשלחים אוטמטית, כמו לדוגמא: הפרמטר event שיש לכל אירוע. אם הפונקציה שלי היא מטפל אירוע (נקראת ע"י שימוש בonXxx) היא תשלח בצורה אוטומטית את המשתנה (המשתנה לא חייב להקרא כך. אם יש פרמטר אחד שהפונקציה מצפה לקבל, ואף אחד לא שולח, פרמטר הevent ישלח. מקובל לקרוא לו e/ev/event)

פרמטר זה מכיל מידע על האירוע שלי, לדוגמא: בעת הקלדה- על המקש שהוקלד כעת, באירועי עכבר: מידע על מיקום העכבר, לחיצה ימנית/שמאלית, מידע על האובייקט שהפעיל את מטפל האירוע ועוד.

לכן, אם הצהרת הפונקציה שלי תהיה function myFunc (e) {...}, ניתן לקרוא כך לפונקציה:

<button onClick={myFunc}> למרות שלכאורה לא נשלח לפונקציה הפרמטר, הוא נשלח בצורה מרומזת.

העברת נתונים לקומפוננטה.

עד עכשיו, כשהצהרנו על פונקציית הקומפוננטה, הסוגריים של הפרמטרים, נשארו ריקים – כלומר הקומפוננטה לא קיבלה שום ערכים מבחוץ.

בבניית אפליקיצית ווב כל שהיא נרצה להעביר נתונים לקומפוננטטה שלנו.

נוכל להשוות זאת למבנה הhtml הרגיל בו אנחנו מבצעים את אותה פעולה – שולחים מאפיינים אותם התגית מעבדת ומכניסה למקומות המתאימים, יוצרת התנהגות מתאימה – לדוגמא href בתגית a.

גם לקומפנטות שלנו נרצה לעיתים להעביר נתונים: לדוגמא, קומפוננטה המייצגת כרטיס לקוח, נרצה להעביר את פרטי הלקוח, או נתונים עבור עיצוב קומפוננטה כמו צבעים וכו'

כדי להשתמש באפשרות זו, יש לבצע את השלבים הבאים:

1. בהצהרת הקומפוננטה, יש לקבל משתנה. מקובל לקרוא למשתנה props. כלומר הצהרת הקומפוננטה תראה כך function MyComponent(props){...}
   הprops הוא אובייקט המכיל את כל הattributes שנשלחו לקומפוננטה כattributes לכל דבר.
2. ב"יצירת מופע" (כאשר מביאים אותה למסך שלנו) לקומפוננטה שלנו, יש לשלוח את המאפיינים הנדרשים, כמו ששולחים לכל אלמנט html אחר, כמו href, width, disabled, value וכו'.
3. בתחילת פונקצית הקומפוננטה, יש לקבל את הprops לתוך משתנה מקומי, בצורה מעין זו: לדוגמא, אם הprops שנשלחו הם id,value, השורה בקומפוננטה תהיה const {id,value}=props

זהו, כעת העברנו את הנתונים מהאב המכיל את הקומפוננטה, לתוך הקומפוננטה.

העברת אוביקט כprops.

לעיתים נרצה להעביר אוביקט שלם כprops לקומפוננטה, למשל, לפי הדוגמא הקודמת, נרצה להעביר אוביקט שלם של לקוח לקומפוננטה, ולא שדה שדה.

ניתן לבצע זאת ע"י "פיצול" האוביקט למאפיין מאפיין באמצעות אופרטור שלוש נקודות(...)(יורחב בהמשך) המאפשר לפצל אוביקטים ומערכים לאברים בודדים, אותו נכניס לתוך סוגרים מסולסלות (צומדיים) כדי להדגיש שמדובר באוביקט.

לדוגמא:

{<Client {...x} />}

ערכי ברירת מחדל לprops

בשימוש בקומפוננטה, אין דרך בה נוכל לוודא שכל מי שמשתמש בקומפוננטה שלנו, מעביר לה את כל הprops אותם היא צריכה, דבר שעלול לעיתים ליצור בעיות כמו גישה למשתנים שהם null, ביצוע חישובים עליהם וכו'. הפתרון לבעיה הוא ליצור defualtProps, כלומר ערכי ברירת מחדל לprops של הקומפוננטה שלי. במידה שישלחו ערכים מתאימים, הם ידרסו את ערכי ברירת המחדל. במידה ולא ישלחו ערכים, הקומפוננטה תשתמש בערכי ברירת המחדל.

אופן השימוש:

MyComponent.defualtProps={

Key:value,

Key2:value

}

לדוגמא:

```
import React from 'react';
export default function Person(props) {
    const { firstName, lastName, age, address } = props;

}
Person.defaultProps = {
    firstName: 'Sara',
    lastName: 'Levi',
    age: 10,
    address: 'Jerusalme'
}
… return <div>
        <Person />
        <Person firstName="Rachel" age="30" />
        <Person lastName="Weiss" address="Tel-Aviv" />
        <Person firstName="Chana" lastName="Kliger" age="5" address="Ze
fat" />
    </div>
```

במקרה הראשון: כל הערכים יהיו מערכי ברירת המחדל

במקרה השני: השם הפרטי והגיל יתמלאו במה שנשלח ואילו שם המשפחה והכתובת בברירת המחדל

במקרה השלישי : להפך

במקרה הרביעי: כל הערכים יתמלאו ממה שנשלח.

אופרטור שלוש נקודות (...)

נושא זה אינו קשור באופן ישיר לריאקט, אלא לJS, ES6 ומעלה, אולם בא לידי שימוש במס' רב של מקרים באפליקציות ריאקט, כפי שנראה בהמשך.

אופרטור זה, מאפשר לפצל מערך או אוביקט לרשימה "יחידנית" של איברי האוביקט/המערך.

אופרטור זה בא לידי שימוש פעמים רבות כאשר רוצים לבצע clone (שכפול) לאוביקט/מערך.

הדרך בה זה מתבצע (לדוגמא על מערכים) הוא באופן הבא:

var arr=[1,2,3];

var arr2=[...arr];

ונסביר:

אופרטור ה-3 נקודות, החליף את arr לrange של ערכים, ולא כמערך. זה בעצם כמו לכתוב [1,2,3]. לאחר שנכניס זאת לסוגרים מרובעים, יווצר לנו פה מערך חדש.

אותו דבר גם באוביקטים, רק יש לשים לב שאוביקטים יש לעטוף בסוגרים מסולסלים (צומדיים) כדרך ההצהרה על אוביקטים. לדוגמא

var a={A=1,B=2};

var b={...a};

שמירת state בקומפוננטה:

מה זה state

אחד הדברים שמאפיינים אפליקציה כל שהיא, היא היכולת לשנות נתונים באופן דינמי במהלך ריצת התוכנית.

עד עכשיו, יכולנו לשנות ערכים של משתנים בתוך הקומפוננטה, אולם לאחר שהדף התרנדר, לא הייתה משמעות לשינוי. לדוגמא, אם הצהרנו שX=5, הצגנו אותו על המסך ובעת לחיצה ערך הX השתנה, התצוגה במסך לא השתנתה.

הדרך לגרום לtwo way binding (תלות דו כיוונית) כלומר ששינוי במשתנים ישפיע על התצוגה באתר וכן להפך הוא שימוש בפונקציה useState.

אופן השימוש בפונקציה:

1. יש לייבא את הרכיב useState מהקובץ react בצורה הזו:

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

יש להכניס את שם הרכיב בתוך סוגריים מסולסלים

2. "ייצור" stateה למשתנה הרצוי: התבנית הקבועה לבצע זאת היא באופן הבא, לדוגמא

const [num,setNum]=useState(0);

הסבר: ב"מערך" (זה לא מערך, הוא רק מוצהר כך.) המוגדר יש להגדיר שתי ערכים.

א. המשתנה עליו נרצה לבצע מניפולציה במהלך ריצת התוכנית

ב. שם פונקציה אשר משנה את המשתנה שהוגדר בסעיף א.

יש לשים לב, שהפונקציה setNum אינה מוצהרת שוב בהמשך, אלא רק כאן באופן מרומז.

את אופן השימוש בפונקציה, נסביר בהמשך.

הפונקציה useState היא פונקציה בreact שמגדירה שמשתנה זה, שהוצהר בסעיף א' ינוהל באמצעות stateה של react. ערך הפרמטר שהפונקציה מקבלת הוא הערך הראשוני של המשתנה. למשל, בדוגמא שלנו, יוצהר מאחורי הקלעים משתנה שנקרא num שהערך הראשוני שלו יהיה 0.

3. עדכון stateה.

יש לשים לב לנקודה חשובה בריאקט שלעולם לא נשנה משתנה שהוצהר בstate באמצעות השמה ישירה (כמו num=1, או num++) שינוי הערכים היא תמיד באמצעות הפונקציה

המרומזת שהוגדרה בעת יצירת הstete כלומר setNum. פונקציה זו מקבלת כפרמטר את הערך החדש עבור המשתנה num לדוגמא 1, או num+1.
חובה! לשים לב לא לשנות את ערך המשתנה ישירות, אלא רק באמצעות הsetSate.

שמירת ועדכון אוביקטים ומערכים בstate.

מכיון שראינו שלא מעדכנים את הערך ישירות, אלא רק באמצעות הsetState, ישנה בעיה לבצע זאת על מערכים ואוביקטים כיוון שלא ניתן לשנות את אוביקט המקור, אלא יש צורך לשלוח אוביקט/מערך חדש עם הערכים המעודכנים.
הפתרון: ליצור העתק לאוביקט, לשנות אותו, ואת העתק לשלוח לפונקציה setState.
שכפול האוביקט נעשה באמצעות אופרטור שלוש נקודות עליו הרחבנו לעיל.
לדוגמא

```
const [client,setClient]=useState({});

function updateClient(){
var cloneClient={...client};
cloneClient.XXX="";
setClient(cloneClient);
}
```

מימוש two way binding בפקדים שונים

עד עכשיו, צורת העדכון היתה one way binding – מקוד הJS לUI.

כפי שנוכל לראות, ישנו צורך גם במימוש two way binding – כלומר שהUI ישפיע על המשתנים השמורים בזכרון. אחד הדוגמאות הנפוצות זוהי תיבת טקסט. אנו נרצה שתוכן הinput בUI ישתנה בהתאם לערך השמור בזכרון בJS. כמו כן, נרצה שאירוע הקלדה מהUI ישפיע על הערך השמור ב-state. הדרך להגדיר זאת היא ע"י שימוש ב two way binding במבנה באופן הבא

כשמו כן הוא two way binding **two**, מגדירים באמצעות שני attributes באלמנטים שלי

א. value ={textVaribleInJs}
ב. onChange={(e)=> setValue(e.target.value)}

הסבר: בחלק הראשון נקשר את הUI לJS, ואילו בחלק השני נעדכן את הJS לאחר אירוע מהUI.

(צורת הכתיבה של הsetValue היא פונקציה אנונימית, שמקבלת את הevent למשתנה e.

למשתנה e יש property בשם target, המכיל את האלמנט שהפעיל את האירוע. לtarget יש property בשם value המכיל את ערך האלמנט (במקרה שלנו – תיבת טקסט). לכן כדי לשלוח לעדכון בstate את הערך החדש, נשתמש בe.target.value)

לדוגמא:

```
const [name,setName]=useState('');

return <div>
    <input value={name} onChange={(e)=>setName(e.target.value)}/>
```

```
</div>
```

נושאים שונים בריאקט

גישה ישירה לאלמנטים בDOM - useRef

ריאקט מאפשר לנו לגשת ישירות לאלמנטים שעל העמוד באמצעות פונקציה בשם useRef .

ככלל, צורת הכתיבה בריאקט לא 'מעודדת' שימוש בגישה ישירה לאלמנטים בDOM, ממס' סיבות כמו לדוגמא הרצון לייצר פקדים שלא בהכרח מחוברים ל DOM וכך אפשר יהיה להשתמש בהם גם בקוד צד שרת או בסביבת React Native, וכו'.

אולם לעיתים ישנו צורך דווקא בגישה ישירה לDOM , ולקבל גישה לאלמנטים בעצמם כמו לדוגמא במקרים הבאים:

1. גישה ישירה לאלמנט לשם הפעלת מתודה על אותו האלמנט (דוגמא בהמשך)
2. ישנן ספריות או קוד ישן, שאופן פעולתם דורש קבלת DOM element
3. לעיתים בעזרת useRef ניתן לכתוב קוד גנרי או מהיר יותר

אין כלל ברור מתי ניתן להשתמש בuseRef, המפתח הוא להפעיל שיקול דעת ולהשתמש ב ref כשחייבים אותו, מתוך הבנה שזו לא הדרך המומלצת והרבה פעמים אם אפשר עדיף לבחור בדרכים אחרות.

נושא אחד שחייבים בשבילו לגשת ישירות לאלמנטים הוא הגדרת focus לאלמנט. הדרך ב DOM לקבוע פוקוס של אלמנט היא להפעיל את המתודה: element.focus()

ולכן אם אנחנו רוצים לשנות פוקוס נצטרך גישה לאלמנט עצמו.

כדי לגשת לאלמנט בריאקט אנחנו עושים שני דברים:

1. אנחנו מפעילים את הפונקציה useRef בתחילת הפקד שלנו כדי לקבל אוביקט הפניה מיוחד.
2. אנחנו מעבירים את אוביקט ההפניה הזה בתור מאפיין ref של Virtual DOM Element.
   מאפיין ref הוא לא מאפיין שממשיך ל HTML, ובדומה ל key הוא מאפיין שנועד רק בשביל ריאקט. ריאקט ימלא את השדה current של אוביקט ה ref שקיבל בהפניה ל DOM Element שנוצר.

לדוגמא: קומפוננטה המכילה שני input-ים, ומגדירה מעבר מהinput הראשון לשני לאחר סיום הקלדת מילה (שימוש בתו רווח (' '))

הפונקציה useRef מקבלת ערך ראשוני של אוביקט הפניה (בדרך כלל נעביר שם null) ומחזירה אויבקט ref מיוחד. את האוביקט שהיא החזירה אנחנו יכולים להעביר כמאפיין ref לכל אלמנט ב JSX.

המשתנה refToWord2 כולל אוביקט ששדה current שלו הוא בדיוק ה DOM Element שמתאים ל input השני בפקד.

הקוד יראה כך:

```
const [word1, setWord1] = useState('');
   const [word2, setWord2] = useState('');
   const refToWord2 = React.createRef();

   function setFirstWord(val) {
```

```
        setWord1(val);
        if (val.endsWith(' ')) {
          refToWord2.current.focus();
        }
      }

    function setSecondWord(val) {
      setWord2(val);
    }


    return (
      <div>
        <input type="text" value={word1} onChange={(e) => setFirstWord(
e.target.value)} />
        <input type="text" value={word2} onChange={(e) => setSecondWord
(e.target.value)} ref={refToWord2} />
      </div>
    )
```

טריגר בעת עדכון state - useEffect

הhook (נושא הhooks יוסבר בהמשך – כרגע פונקציה) useEffect משמש ל"מעקב" על שינויים בstate מסוים.

ייעוד נוסף שלו הוא הפעלת פונקציה בעת עליה ראשונה של קומפוננטה או בכל רינדור.

הפונקציה מקבלת שני פרמטרים:

1. effect – פונקציית callback- מה לעשות כאשר->
2. deps (אופציונלי) – רשימת תלויות- מבצע את הפונקציה שנשלחה בפרמטר הראשון כאשר אחד המשתנים במערך התעדכנו.

לדוגמא:

```
import React, { useEffect, useState } from 'react';
export default function All(props) {
    const [userName, setUserName] = useState('');
    useEffect(function () {console.log(userName);},
     [userName]);
    return <><input type="text" value={userName} onChange={(e) => setUs
erName(e.target.value)} /></>;
}
```

נוכל לראות בדוגמא זו, שהמעקב מתבצע אחרי משתנה הstate -username. (נשלח כאיבר במערך התלויות – הפרמטר השני שנשלח לפונקציה useEffect. כאשר יתבצע שינוי כל שהוא במשתנה שבstate, תופעל הפונקציה שנשלחה כפרמטר הראשון, ויודפס ללוג ערך המשתנה.

שימו לב! הפונקציה תיקרא גם אם ערך המשתנה יתעדכן בעקבות שינוי דרך הJS, ולא דווקא ע"י אירוע משתמש (כמו הקלדה) (לדוגמא, תוצאת חישוב וכו')

כאשר רוצים להפעיל פונקציה מסוימת בכל פעם שהקומפוננטה מרנדרדת את עצמה, יש לקרוא לפונקציה useEffect ולשלוח לה את הפרמטר הראשון בלבד. את הפרמטר השני, שהוא אופציונלי, אין לשלוח (גם לא כמערך ריק!)

לדוגמא:

```
useEffect(function(params) {console.log("component rendered")});
```

במידה שרוצים להפעיל קטע קוד מסוים פעם אחת בלבד, בעת עלית הקומפוננטה, יש לקרוא לפונקציה useEffect, ולרשימת הDepends יש לשלוח מערך ריק. שימוש נפוץ בצורה זו, הוא למטרת הפעלת timer בקומפוננטה. חשוב להבהיר. במידה ורוצים להשתמש בפונקציה setInterval של JS- (הפעלת טיימר) יש לוודא שהיא תקרא פעם אחת בלבד, לכן ניתן לקרוא לה כתגובה לאירוע מהמשתמש (לדוגמא – לחיצה על כפתור) ואז לוודא שלא ניתן יהיה להפעיל שוב, או, במקרה הנפוץ יותר – להפעיל בפעם הראשונה שהקומפוננטה נטענת:

לדוגמא:

```
const [second,setSecond]=useState(0);
useEffect(function(params) {
    const interval=setInterval(function(params) {
        setSecond(second=>second+1);
    })
},[]);
```

כתיבת קומפוננטה כמחלקה:

בראשית דרכה, ריאקט התחילה את מבנה הקומפוננטות כמחלקות. כיום מקובל יותר לכתוב "functional component" ולא "class componenent", אולם עדין ניתן לראות קומפוננטות הכתובות כמחלקה.

בפרק זה נסביר את עיקרי ההבדלים בין שתי הדרכים לכתיבת קומפוננטות.

1. הצהרת הקומפוננטה – במקום **function** export default נכתוב export default **class XXX extends React.Component**
2. ערך מוחזר: בשונה מפונקציה במחלקה לא ניתן לבצע return, לכן יש לממש את הפונקציה **render** בכל קומפוננטה שהיא מחלקה. פונקציה זו תחזיר את הJSX הדרוש עבור הקומפוננטה.
3. קבלת הprops. מכיון שלמחלקה לא ניתן לשלוח פרמטרים כמו לפונקציה, הדרך להעביר את הprops היא באמצעות הconstructor של המחלקה במבנה הבא: constructor(props){...} השורה הראשונה בפונקציה הבונה תהייה קריאה לבונה במחלקת האב באמצעות super(props)
4. איתחול ועדכון הstate: בקומפוננטת מחלקה – לא ניתן להשתמש בuseState. הדרך לשמור ולעדכן את הstate היא באופן הבא:
בבונה של המחלקה, יש לאתחל את הstate המקורי – אובייקט ברמת המחלקה המכיל את כל משתני הstate של הקומפוננטה, בערכי ברירת המחדל. לדוגמא:
this.state = {firstName:'',lastName:'' };
כדי לעדכן את הstate, יש ליצור פונקצית עדכון (פונקציה שבתוכה נקרא לפונקציה setState- מובנית גם היא בקומפוננטות בכתיב מחלקה), ולאחר מכן יש לבצע bind.
את ה bind יש לבצע לאחר איתחול הstate- בבונה. לדוגמא:
constructor(props)}

```
        super(props);
        this.state = {firstName:'',lastName:'' };
        this.changeData = this.changeData.bind(this);
      {

       changeData(prop,value)}
          var cloneObj={...this.state};
          cloneObj[prop]=value
        this.setState( cloneObj);
       }
```

דוגמא למחלקה שלמה:

```
export default class Person extends React.Component {
  constructor(props) {
    super(props);
    var {age}=props;
    this.state = {firstName:'',lastName:'',age };
    this.changeData = this.changeData.bind(this);
  }

  changeData(prop,value) {
      var cloneObj={...this.state};
      cloneObj[prop]=value
    this.setState( cloneObj);
  }

  render() {
    const { firstName,lastName,age } = this.state;
    return (
      <p style={style}>Hello. My name is {firstName +" "+lastName} and
I'm {age} years old</p>
    );
  }
}
```

אופן השימוש בקומפוננטה מסוג זה, **זהה לחלוטין** לשימוש בקומפוננטת פונקציה:

```
<Person age="12"/>
```

זרימת מידע בריאקט

שיתוף המידע בריאקט הוא אחד העקרונות החשובים של הספריה והוא זה שעוזר לנו לקבל ממשקי משתמש מורכבים בלי טעויות.

מי אחראי על המידע

כל הפקדים שראינו עד עכשיו החזיקו את ה state שלהם אצלם וכל פעם שקרה משהו בפקד ה state שלו השתנה ופונקציית render של הפקד שיקפה את השינוי למשתמש. למעשה בעולם האמיתי יש לנו כל הזמן קשר בין מספר פקדים:

לדוגמא: כפתור שמירה הופך ל Enabled רק אחרי שמשתמש הכניס ערכים תקניים בכל השדות הנדרשים.

שם המשתמש מוצג במס' דפים באתר וכו'

מאחר שאנחנו לא רוצים לבנות את כל המערכת שלנו בתור פקד יחיד, עלינו למצוא דרך לשתף את המידע בין מספר פקדים.

בריאקט בכל מקרה של שיתוף מידע יש לנו קומפוננטה אחת ראשית שיצרה את המידע באמצעות useState. זו הקומפוננטה שאחראית לשמור ולעדכן שדה מידע זה. הקומפוננטות הפנימיות יותר יוכלו לשמור מידע בעצמן באמצעות useState שהן יפעילו, אבל המידע שמשותף חייב להישמר בקומפוננטה העליונה ביותר שצריכה אותו. מידע בריאקט עובר מלמעלה למטה.

לדוגמא פקד של מונה לחיצות שרוצה להיעזר בפקד נוסף כדי להציג את מספר הפעמים שמשתמש לחץ על הכפתור: במצב כזה המידע עצמו יישמר בפקד מונה הלחיצות, ויהיה פקד פנימי יותר שאחראי על הצגת המספר. בקוד זה עשוי להיראות כך:

```
function Display(props) {
    return (
      <div>You scored ... points</div>
    );
  }

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

    function inc() {
      setCount(x => x + 1);
    }

    return (
      <div>
        <Display />
        <button onClick={inc}>Click Me</button>
      </div>
    );
  }
```

שליחת מידע לילדים

כמובן בשביל ש Display יציג את מספר הלחיצות אנחנו צריכים להעביר את המספר הזה באיזשהו אופן. לכן עלינו לשלוח את המספר בתור Property מ Counter ל Display. הקוד יראה כך:

```
  ...

return (
      <div>
        <Display score={count} />
        <button onClick={inc}>Click Me</button>
      </div>
```

```
    );
  …
```

שליחת פונקצית עדכון לילדים

יתכנו מצבים בהם נצטרך לאפשר לילדים לשנות את המידע בעצמם? מה יקרה למשל אם בתוך אלמנט Score נרצה להוסיף כפתור Reset שמאפס את הנקודות?

במצבים כאלה אנחנו יוצרים פונקציה חדשה בפקד שמנהל את המידע, ומעבירים את הפונקציה עצמה בתור Property לילדים שצריכים לשנות את הערך. בחזרה לדוגמת מונה הלחיצות וכפתור ה Reset, הקוד יראה כך:

```
function Display(props) {
    const { score, reset } = props;
    return (
      <div>
        You scored {score} points. <button onClick={reset}>Reset</butto
n>
      </div>
    );
  }

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

    function inc() {
      setCount(x => x + 1);
    }

    function reset() {
      setCount(0);
    }

    return (
      <div>
        <Display score={count} reset={reset} />
        <button onClick={inc}>Click Me</button>
      </div>
    );
  }
```

ניתן להעביר כל סוג של מידע בתור Attribute השיטה תמשיך לעבוד גם אם Display יחזיק היררכיה של פקדים פנימיים וימשיך לשלוח אותה למטה בשרשרת. זה לא משנה מי יפעיל את reset, מה שחשוב זה שהיא נוצרה אצל הפקד ששומר את המידע.

איך לשתף מידע בין אחים

שיתוף מידע בריאקט הוא תמיד בצורת מפל: המידע עובר מפקד-אב שמחזיק את המידע לילדים שלו. לכן כשאנחנו רוצים לשתף מידע בין פקדים שנמצאים באותה רמה הדרך המקובלת היא ליצור פקד חדש מעל שניהם שינהל את המידע המשותף.

Redux- פיתוח יישומים גדולים

למה לא לשמור מידע בפקד

אנחנו יודעים לשמור מידע בתור משתנה state של פקד, וגם להעביר מידע מפקד לילדים שלו, אבל העסק הזה הופך קשה יותר ויותר ככל שהמרחק בין הפקד העליון ביותר שצריך את המידע לפקד התחתון ביותר שצריך אותו גדל. לדוגמא, אם המערכת שלנו שומרת את המשתמש שמחובר, ואנחנו נרצה להשתמש במידע זה מהרבה מאוד פקדים במערכת, נצטרך לשמור את המידע בסטייט של הפקד העליון ביותר ולהעביר אותו כלפי מטה לפעמים 4-5 פקדים פנימה רק כדי שפקד פנימי כל שהוא יוכל להציג את שם המשתמש או להציג UI אחר אם יש או אין משתמש מחובר.

אתגר נוסף של שמירת מידע רק בתוך הפקדים הוא הבדיקות - הרבה פעמים יש לוגיקה שניתן לבדוק אותה בלי קשר ל UI, אבל אם כל המידע נשמר בפקדים אין אפשרות לכתוב בדיקות נקיות רק של הלוגיקה וחייב את כל פקדי ריאקט על המסך בשביל לבדוק.

לכן יהיה לנו הרבה פעמים מאוד נוח לשמור את המידע והלוגיקה של היישום מחוץ לפקדים, ולהשתמש בפקדים רק בתור שכבת UI שמקבלת את המידע מבחוץ ומציגה אותו. רעיון זה הביא לארכיטקטורה שנקראת Flux.

מהי ארכיטקטורת Flux

ארכיטקטורת Flux בנויה על מספר רעיונות מרכזיים:

הרכיב המרכזי נקרא Store. מעין מחסן מידע חיצוני לפקדים שתפקידו לשמור את המידע הגלובאלי של היישום. בFlux הקלאסי יהיה Store לכל יישות ביישום. כל קומפוננטה יכולה לקרוא מידע מכל Store, ובאופן אוטומטי כשמשהו ב Store משתנה כל הקומפוננטות שקשורות אליו ירונדרו מחדש.

כדי להודיע ל Store שמשהו קרה קומפוננטה שולחת Action. ה Action הוא אוביקט ותפקידו של ה Store להחליט איך לטפל ב Action הזה.

בגלל שקומפוננטה לא יודעת איזה Store יצטרך לטפל באיזה Action, יש לנו רכיב נוסף שנקרא Dispatcher. ה Dispatcher תופס את ה Action ומעביר אותו ל Store שמחכה לו.

תהליך זרימת המידע יהיה כך:

1. המידע הראשוני נכנס ל Stores וגורם להצגת ממשק משתמש ראשוני על המסך.
2. משתמש מבצע פעולה ואז הקומפוננטה הרלוונטית זורקת Action ל Dispatcher.
3. ה Dispatcher מפנה את ה Action לכל ה Stores הרלוונטיים, וכל Store שמאזין ל- Action מתעדכן.
4. בעקבות העדכון במידע ב Stores גם הקומפוננטות שמקשיבות ל Stores אלה מרונדרות מחדש.

מימוש הFlux- בריאקט – Redux

בreduxיש רק Store אחד. כל המידע שנשמר ב Store הוא Immutable.

בכל פעם שה Store משתנה מכל סיבה שהיא, כל הקומפוננטות במסך ירונדרו מחדש.

מכיון שכל המידע הוא Immutable, באופן אוטומטי נוכל לדלג על Render של קומפוננטות אם המידע שהן תלויות בו לא משתנה.

מהסיבה שיש רק Store אחד, ה Store הוא גם ה Dispatcher. הוא מקבל את כל ה Actions ומעדכן את עצמו בהתאם – ההרחבה בהמשך.

עבודה עם Immutable Data

אחד העקרונות החשובים בredux, הוא עבודה עם Immutable Data - במבני נתונים שאינם משתנים

זהו הבסיס לזיהוי השינויים האוטומטי שמאפשר לדלג על render-ים.

כדי להקל את כתיבת הקוד כ-Immutable Data , נשתמש בספריה immer.

כדי להשתמש בimmer יש להתקין את הספריה באמצעות הפקודה npm install immer

למה לא ניתן להשתמש ב JS פשוט?

הבעיה עם JavaScript היא שמבני הנתונים הבסיסיים של השפה הם Mutable, כלומר אנחנו יכולים לשנות אותם אחרי שיצרנו אותם. אם בכל זאת נרצה לעבוד איתם בלי לשנות אותם נצטרך לכתוב המון קוד עוטף.

בעבודה עם redux שמחזיק את כל המידע של היישום שלנו בתוך אוביקט יחיד בזיכרון, זה מקשה מאד על כתיבת הקוד

נניח שנרצה לשמור בstore שלנו יישום של צ'אט. אובייקט המידע עשוי להיראות כך:

```
const state = {
        user: 'Leah',
        messages: [
          { from: 'Sara', text: 'Hello!' },
          { from: 'Rivka', text: 'Good morning' },
          { from:'Rachel', text: 'Good night' },
        ],
      };
```

ואנחנו רוצים לשנות את הטקסט של ההודעה האחרונה ברשימה. ב JavaScript רגיל ניתן היה לכתוב בקלות:

```
state[messages][2].text = 'Good Luck';
```

וזה היה עובד. אבל אם משתנה זה היה שמור ב State של פקד אז ריאקט לא היתה מזהה את השינוי ולא היתה מבצעת render מחדש לפקד. גם בעבודה עם redux אנחנו חייבים לעבוד עם Immutable Data כי כל המידע שיישמר ב Store חייב להיות Immutable.

מידע שהוא Immutable אומר שאנחנו צריכים לבצע פעולה שאחריה:

1. המשתנה state יחזיק ערך חדש. לא רק הערכים אלא האובייקט עצמו כלומר יצביע על מקום אחר בזיכרון.
2. המשתנה state.user יצביע כן על אותו ערך שהיה לו לפני השינוי, כי חלק זה באוביקט לא השתנה.
3. המשתנה state.messages יצביע על מערך חדש, כיוון שיש שינוי ברשימת ההודעות.
4. המשתנה state.messages[0] יצביע על אותו אוביקט שהוא הצביע עליו קודם, כיוון שבהודעה זו לא היה שינוי.
5. המשתנה state.messages[1] יצביע על אותו אוביקט שהוא התיחס אליו קודם, כיוון שגם פה לא היה שינוי.
6. המשתנה state.messages[2] יכיל ערך חדש כיוון שבהודעה זו יש שינוי.

בקוד אפשר לכתוב את הסעיפים האלה כך:

```
const state = {
       user: 'Leah',
       messages: [
         { from: 'Sara', text: 'Hello!' },
         { from: 'Rivka', text: 'Good morning' },
         { from:'Rachel', text: 'Good night' },
       ],
     };

     const newState = changeTextOfLastMessageInAnImmutableWay(state);
     console.log(newState);
     console.log(state !== newState);
     console.log(state.user === newState.user);
     console.log(state.messages !== newState.messages);
     console.log(state.messages[0] === newState.messages[0]);
     console.log(state.messages[1] === newState.messages[1]);
     console.log(state.messages[2] !== newState.messages[2]);
     console.log(state.messages[2].from == newState.messages[2].from);
     console.log(state.messages[2].text != newState.messages[2].text);
```

כעת המטרה היא לממש את changeTextOfLastMessageInAnImmutableWay כדי שכל התנאים יתקיימו כפי שרצינו.

איך ניתן לממש זאת באמצעות immer

הספריה Immer מספקת פונקציה פשוטה שהופכת את JavaScript לשפה שתומכת ב Immutable Data. הפונקציה שנקראת produce מקבלת את הstate המקורי ואובייקט טיוטה. בתוכה כל פעולה שנעשה תתורגם אוטומטית לפעולת Immutable. הפונקציה תחזיר אוביקט חדש שבו המצביעים לשדות שהשתנו ולparents שלהם יהיו שונים, ואילו כל השאר, ישארו זהים.

לדוגמא, מימוש הפונקציה יהיה:

```
function changeTextOfLastMessageInAnImmutableWay(state) {
        return produce(state, draft => {
```

```
        draft.messages[2].text = 'Good Luck';
      });
    }
```

במקרה הזה, יוחזר אוביקט, שמצביעיו זהים לדוגמא שיצרנו קודם.

ניתן "לשכלל" את הפונקציה באמצעות פיצ'ר נוסף: אם נעביר לפונקציה produce כפרמטר ראשון פונקציה עדכון ה-state, הפונקציה produce תחזיר פונקציה שמקבלת כפרמטר את הדבר שאנחנו רוצים לשנות (state) ובאופן אוטומטי הופכת אותו ל Immutable Data. לדוגמא:

```
const changeTextOfLastMessageInAnImmutableWay = produce(draft => {
        draft.messages[2].text = 'Good Luck';
      });
```

כמובן שניתן להגדיר את הפונקציה הפנימית, כך שתקבל יותר פרמטרים. הפרמטרים האלו ישתקפו חזרה בפונקציה שהproduce תחזיר, וכך תיצור לנו את האפשרות לשלוח מבחוץ, את תוכן ההודעה, למשל, או כל פרמטר אחר אותו נרצה לעדכן בstate) . (אפשרות זו תבוא לידי שימוש במס' רב של מקרים בשימוש בredux) לדוגמא:

```
  const setLastMessageText = produce((draft, newText) => {
        const lastIndex = draft.messages.length - 1;

        draft.messages[lastIndex].text = newText;
      });

      const newState = setLastMessageText(state, 'Good Luck');
```

תחילת השימוש בRedux

קצת הסבר על Redux

ננסה לחשוב על היישום שלנו, כרגע ללא הUI, אלא רק על מבנה הנתונים של היישום.

לדוגמא, ביישום של ניהול חנות, נרצה להחזיק את רשימת המחלקות, ובכל מחלקה את רשימת המוצרים. כמו כן, נרצה להחזיק את רשימת העובדים, את שעות הפתיחה, משתמש נוכחי באתר וכו'.

תכנון מבנה הנתונים הוא החלטה לוגית שתשפיע על כל מבנה היישום (ואתה גם כל החלטה אחרת על מבנה אובייקט המידע הזה). ככל שהיישום יותר גדל, יתכנו בעיות שונות ביצירת מבנה הנתונים.

הדבר השני שצריך להחליט עליו לפני שמתחילים לכתוב את הקוד הוא הפעולות שנרצה לבצע על המידע, ועל איזה חלק מהמידע כל פעולה תשפיע. למשל ביישום של החנות נוכל להגדיר את הפעולות "הוספת מחלקה", "הוספת מוצר למחלקה", "עדכון מוצר/מחלקה" ו"מחיקת מוצר/מחלקה", "הוספת/עריכת/מחיקת עובד" וכו'. בRedux פעולה היא גם עדכון נתונים בstate, כמו כל הפעולות לדוגמא שהובאו, ולא רק תגובה לאירוע משתמש.

לאחר מכן נחלק את הפעולות לחלקים בעץ המידע עליו הן יכולות להשפיע. לדוגמא "עדכון רשימת מוצרים במחלקה" למשל תשפיע על השדה departments באובייקט המידע, וכך גם "הוספת מחלקה". לעומתן הפעולות "הוספת עובד", "עריכת שעות עבודה לעובד" וכו' ישפיעו על השדה workers באובייקט המידע.

אז איך כותבים את הקוד

אחרי שהחלטנו על מבנה המידע והפעולות אפשר להתחיל להשתמש בredux כדי לבנות את קוד התוכנית. נגדיר אילו פעולות ישנו את הstate, ואילו מהם, לדוגמא, ישלחו לserver או יבצעו כל פעולה אחרת.

דוגמא לפעולות (הוספת מוצר למחלקה, הוספת מחלקה) (פונקציות אלו יכתבו לרוב בקובץ actions.js):

```
 export function addProductToDepartment(departmentId,product) {
        return { type: 'ADD_PRODUCT_TO_DEPARTMENT', payload: {departmen
tId:departmentId,product:product} };
      }



   export function addDepartment(department) {
        return { type: 'ADD_DEPARTMENT', payload: department };
      }
```

פעולה(action) ב Redux היא אוביקט שמגדיר אירוע ביישום שלנו שמשפיע על מצב היישום. לדוגמא אחרי שנקבל פעולת addDepartment השדה של departmentList ישתנה ויכיל ערך נוסף: הערך שמתואר באירוע. את הדרך בה כל פעולה משפיעה על אוביקט הstateב Redux אפשר לתאר באמצעות פונקציה - פונקציה שתקבל כפרמטר את הstate ואת הפעולה ותחזיר את הstate החדש. הפונקציה בreduxנקראת reducer.

כדי לדעת איזו פעולה נבצע, נסתכל הtypeשבactionשלנו. הוא מגדיר את הפעולה אותה יש לבצע.

כך יראה לדוגמא הreducer ביישום שלנו:

```
const reducer = produce((state, action) => {
    switch(action.type) {
      case 'ADD_PRODUCT_TO_DEPARTMENT':
        state.departments.filter(x=>x.id==action.payload.departmentId).
push(action.payload.product);
        break;

      case 'ADD_DEPARTMENT':
        state.departments.push(action.payload);
        break;
    }
  }, initialState);
```

כאשר הinitialState הוא הstate הראשוני שלנו.

בעצם תיארנו את כל מה שיכול לקרות באפליקציה, קודם כל בצורת אוביקט שמתאר את האירוע ולאחר מכן בתוך הפונקציה שמטפלת באירוע ומראה איך הדבר הזה שקרה משפיע על ה State.

הוספת הreduxלקוד

ראשית, כדי לעבוד עם הredux יש להתקין שתי ספריות

npm install react

npm install react-redux

ניצור קובץ בשם Store.js , ובו נאתחל את הreducer

לאחר מכן ניצור בקובץ הStore שלנו את משתנה הstore באמצעות הפונקציה createStore אותו נייבא מהספריה redux.

הפונקציה מקבלת כפרמטר את הreducer אותו יצרנו בשלב הקודם. את המשתנה store יש להחצין באמצעות השימוש בexport default .

לדוגמא:

```
const Store = createStore(reducer);
export default Store;
```

בנוסף, ניצור קובץ בשם Action.js ולתוכו נכניס את כל הפעולות אותן נרצה לבצע ביישום. (רק את **הפעולות** ולא את המימוש שלהם) (כפי הדוגמאות לעיל) (הסבר על מטרת קובץ הactions, מעט יותר בהמשך)

האובייקט store שנוצר לי באמצעות השימוש בפונקציה createStore מכיל, מלבד הstate כמובן, שתי פונקציות חשובות למהלך השימוש בredux.

1. dispatch – שליחת הודעה לreducer. באמצעות פונקציה זו (אופן השימוש ודוגמאות – בהמשך) נוכל לשלוח הודעת עדכון הstate לreducer שלנו. הפרמטר שישלח הוא action, שמכיל type- כלומר **איזה** עדכון לבצע, ו**payload** – **הנתונים** שפונקצית העדכון צריכה. (פירוט בהמשך)
2. getState – פונקציה שמחזירה את הstate המעודכן, בכל רגע נתון.

בשתי פונקציות אלו, לרוב לא נשתמש באופן ישיר אלא באופן עקיף למעט מקרים בודדים (כמו למשל בmiddlewares, בהמשך) (לדוגמא, קומפוננטה שמשתמשת בstore, תדאג לקרוא לפונקציה getState, כדי להשאיר את הנתונים בקומפוננטה מעודכנת, וכו'.)

העברת הStore באמצעות הProvider

כדי שנוכל להתחיל לכתוב ממשק משתמש ביישום שלנו באמצעות הredux, נעטוף את האלמנט הראשי ביישום (בד"כ בקומפוננטה App) באלמנט Provider מהספריה react-redux.

ל Provider אנחנו צריכים להעביר כפרמטר את ה store שיצרנו ב Redux. בקוד זה נראה כך:

```
import { Provider } from 'react-redux';
import store from './store';

function App (props) {
 return <Provider store={store}>
    <p>Hello World</p>
  </Provider>;
};
```

עכשיו נוכל להעביר את הStore שלנו הלאה, לכל הקומפוננטות:

חיבור הקומפוננטות לStore

למעשה, מה שבצענו עד עכשיו, זה לגלגל את הStore, למטה, לילדים, אולם, זה לא מספיק.

גם הילדים צריכים לדעת "לקבל" את הStore.

לדוגמא

```
import React from 'react';
import { connect } from 'react-redux';

function mapStateToProps(state) {
  return {
    username: state.username,
  };
}

export default connect(mapStateToProps)(function Username(props) {
  const { username } = props;

  return (
    <div className='username'>
      <label>
        User Name:
        <input type="text" value={username} readOnly={true} />
      </label>
    </div>
  );
});
```

ההסבר:

1. נקודת הכניסה היא הפונקציה connect. פונקציה זו היא Higher Order Component שמחברת את הפקד שאנחנו נותנים לה ל Redux Store. הפונקציה מקבלת כפרמטר פונקציה אחרת שבדרך כלל נקרא לה mapStateToProps.
2. פונקציית mapStateToProps אחראית על החיבור בין הState לבין הפקד שלנו: היא בעצם שולפת מידע מהState ומייצרת אוביקט חדש רק עם המידע שרלוונטי לפקד הנוכחי. כל שינוי במידע זה יגרום ל render חדש של הפקד.
3. מנקודת המבט של הפקד כל המפתחות של האוביקט ש mapStateToProps מחזירה עוברים אליו בתור props. כששדות אלה בstate ישתנו באופן אוטומטי יהיה render חדש עם הערכים החדשים.

קריאה לפונקציות העדכון בstore באמצעות dispatch

כדי לקרוא לפונקציות העדכון בreducer שבתוך הstore שלנו, יש להשתמש בפונקציה dispatch.

הפונקציה dispatch שהיא חלק מהstore מקבלת action, כפי שראינו בקובץ הactions

הaction הוא בסה"כ אובייקט שמכיל שני מאפיינים (אותם אובייקטים שמחזירות לנו הפונקציות בקובץ actions) type ו-payload .

הtype הוא הפעולה אותה נרצה לבצע

הpayload יכיל את הנתונים אותם פונקצית העדכון צריכה לצורך העדכון, לדוגמא: מוצר חדש, שם משתמש חדש, מזהה מחלקה למחיקה וכו'.

למעשה קובץ הactions נועד להקל על שליחת אירועי הdispatch.

במקום כל פעם לייצר מחדש את הaction שלי, ולכתוב שוב במס' מקומות את הקבוע של הtype (שהוא string המגדיר את הפעולה אותה יש לבצע), ולהחזיר בpayload את האובייקט המבוקש, הפונקציות בactions.js תקבלנה את הpayload בלבד, ותחזרנה action המורכב גם מה payload שהתקבל כפרמטר, וגם מהtype. לפי הtype, נבדוק בreducer שלנו (באמצעות הswitch case) איזו פעולה יש לבצע עכשיו.

את הפונקציה dispatch נקבל כחלק מהprops בכל קומפוננטה אותה נעטוף בconnect.

לדוגמא:

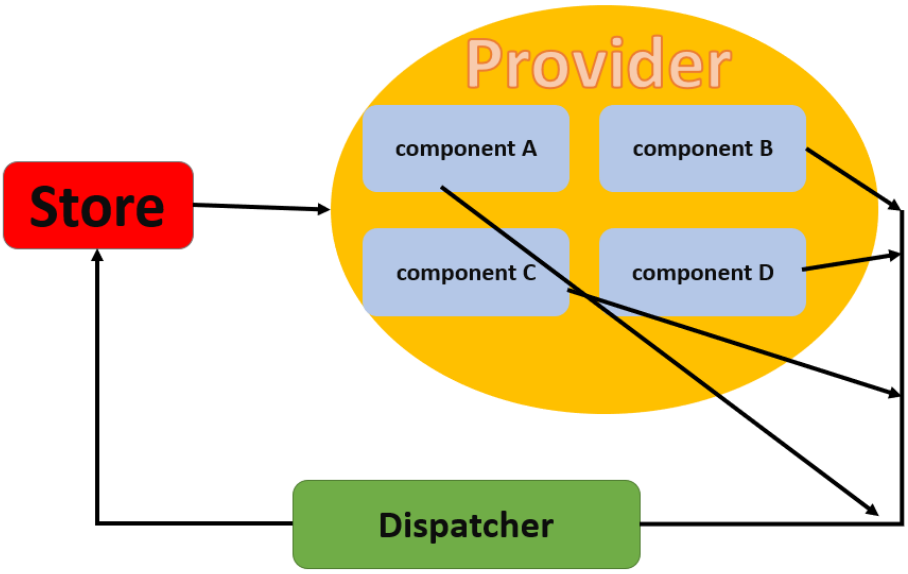
```
import React from 'react';
import { connect } from 'react-redux';

function mapStateToProps(state) {
  return {
    username: state.username,
  };
}

export default connect(mapStateToProps)(function Username(props) {
    const { username, dispatch } = props;

    function handleChange(e) {
      dispatch(setUsername(e.target.value));
    }

    return (
      <div className='username'>
        <label>
          User Name:
          <input type="text" value={username} onChange={handleChange} /
>
        </label>
      </div>
    );
  });
```

(את הפונקציה setUserName נביא מקובץ הActions שלנו.)

מה שיקרה בקומפוננטה הוא כך:

כאשר הקומפוננטה נטענת, ערך שדה הטקסט יכיל את הערך הראשוני של השדה username בstore. בכל הקלדה, תקרא כמטפל(ת) אירוע הפונקציה handleChange. הפונקציה תשלח עדכון לStore באמצעות הפונקציה dispatch. הreducer שלנו יעדכן את הstate, וכתוצאה מכך שוב יתעדכן הuserName בקומפוננטה וחוזר חלילה.

ניתן לתאר את מחזור החיים של פקדים בריאקט באופן הבא:



פיצול אוביקט המידע ביישום Redux

למה לפצל?

הדוגמאות שראינו עד עכשיו היו יחסית קטנות, ואחת השאלות הראשונות שעולות כשפוגשים את Redux לראשונה היא איך זה עובד ביישומים גדולים יותר, כשיש המון מידע שצריך לשמור.

נקח שוב כדוגמא יישום המתאר חדר בצ'אט.

זה ה state הראשוני:

```
const initialState = {
    rooms: [
      { id: 0, name: 'חדר1' },
      { id: 1, name: 'חדר2' },
    ],
    activeRoomId: 0,
    messages: [
      { id: 0, from: 'שרה', text: 'שלום' },
    ],
    username: "אורח",
  };
```

נוסיף אליו את הreducer שלנו:

```
const reducer = produce((state, action) => {
```

```
    switch(action.type) {
      case 'SET_USERNAME':
        state.username = action.payload;
        break;

      case 'RECEIVED_MESSAGE':
        state.messages.push(action.payload);
        break;

      case 'CREATE_ROOM':
        state.rooms.push({ id: nextId(state.rooms), name: action.payloa
d });
        break;

      case 'SET_ACTIVE_ROOM':
        state.activeRoomId = action.payload;
        break;

      case 'RECEIVED_ROOMS':
        state.rooms = action.payload;
        break;
    }
  }, initialState);
```

ניתן לראות שהקוד מטפל במספר נושאים יחד:

1. הקוד מטפל בכל מנגנון ההודעות, בכניסה של הודעות חדשות למערכת והצגתן.
2. הקוד מטפל בכל מנגנון החדרים, בקבלת רשימת חדרים חדשה ובמעבר לחדר אחר.
3. הקוד מטפל בנושא בחירת שם המשתמש.

ביישום אמיתי יתכן שכל היבט של הפונקציונאליות ישפיע על פקדים אחרים או אפילו על דפים אחרים באפליקציה. אפשר לדמיין אפליקציית צ'אט בה במסך הכניסה בוחרים שם משתמש, אחרי זה בוחרים חדר מתוך רשימת חדרים ואז בכניסה לחדר רואים את כל ההודעות שבחדר. במצב כזה וככל שנוצרים לנו יותר אזורים עצמאיים באפליקציה כדאי לפצל את אוביקט המידע ואת ה Reducers שלו לפי אותם אזורים.

כיצד מפצלים?

הצעד הראשון בפיצול הוא לזהות את האזורים השונים באפליקציה ולהפריד כל אזור לאוביקט מידע שלו. לדוגמא, ביישום שלנו, נפצל את האוביקט ל-3 אזורים:

1. הקוד שמטפל בהודעות יהיה באזור אחד שנקרא messages
2. הקוד שמטפל בחדרים יהיה באזור שני שנקרא rooms
3. הקוד שמטפל בפרטי החיבור ושם המשתמש יהיה באזור שלישי שנקרא account

אוביקט המידע אחרי השינוי יראה כך:

```
const initialState = {
    rooms: {
        rooms: [
            { id: 0, name: 'חדר1' },
            { id: 1, name: 'חדר2' },
        ],
        activeRoomId: 0
    },
    messages: {
        messages: [
            { id: 0, from: 'שרה', text: 'שלום' },
        ]
    },
    account:{username: "אורח"}
};
```

החלוקה לאזורים מאפשרת לעשות שינוי מעניין בקוד ה reducer: במקום להשתמש בפונקציה אחת שתמיד יודעת מה לעשות, נוכל לכתוב פונקציה שתמיד יוצרת אוביקט חדש אבל נעזרת בפונקציות נוספות כדי לדעת איך האירוע ישפיע על האזורים השונים באוביקט החדש. בקוד זה אומר שה reducerשלנו יראה בערך כך:

```
function reducer(state, action) {
    return {
      rooms: roomsReducer(state.rooms, action),
      messages: messagesReducer(state.messages, action),
      account: accountReducer(state.account, action),
    }
  }
```

בעבודה רגילה על המערכת בכל פעם שנרצה להוסיף אזור חדש נצטרך להוסיף Reducer חדש ולהוסיף אותו לרשימה כאן, וכל פעם שנרצה לשנות התנהגות של אזור מסוים נוכל ללכת לקובץ בו מוגדר אותו אזור.

כדי לקצר את התהליך, לReduxיש כבר פונקציה בשם combineReducers שמייצרת בדיוק את פונקציית ה reducer שכתבנו בצורה יחסית אוטומטית, כלומר ניתן לכתוב:

```
const reducer = combineReducers({ rooms, messages, account });
```

יש ל**שים לב**, שכל אירוע יגיע לכל הreducer! במידה שמוגדר בו טיפול, האירוע יטופל.

Redux Middlewares

מה הצורך בmiddlewares

בעבודה עם Reducers הדינמיקה של היישום היתה מאוד פשוטה:

קומפוננטה(או כל רכיב אחר) זורקת אירוע באמצעות dispatch

האירוע, שמיוצג בתור אוביקט עם type ו payload, נשלח ל Reducer - ודרכו לכל ה Reducers במערכת שמשנים את הstate של היישום.

בעקבות שינוי הstate כל הפקדים עוברים Render (רק פקדים שמשהו באמת השתנה בפרמטרים שלהם באמת יעברו את הרנדר.)

אולם, תתכנה שתי מגבלות מרכזיות:

1. ה Reducer חייב להחזיר ערך, אבל לא כל פעולה מחזירה ערך. כמו כן, לא ניתן לטפל בפונקציה א-סינכורנית
2. קשה לשתף קוד בין מספר Reducers או בין Reducers בפרויקטים שונים.

דרך אחת לפתור את שתי הבעיות נקראת Redux Middleware. הmiddleware הוא רכיב שיושב לפני ה Reducer ויכול לבצע שינויים עם האירועים שמגיעים ל Dispatch. לדוגמא:

1. Middlewareיכול לכתוב הודעת לוג כל פעם שנכנס אירוע מסוים
2. middleware יכול לזרוק אירועים מסוג מסוים לפח ולא להעביר אותם ל Reducer
3. middleware יכול להפוך אירוע בודד למספר אירועים ולשלוח אותם ל Reducer אחד אחרי השני או אפילו בצורה א-סינכרונית.

הmiddleware נכתב בתור פונקציה נפרדת ואפשר להשתמש בו במספר פרויקטים. בנוסף כל Redux Store יכול להחזיק מספר Middlewares.

כיצד כותבים middleware

middleware הוא פונקציה שמקבלת כפרמטרים את אוביקט ה store ומחזירה פונקציה חדשה, הפונקציה החדשה מקבלת כפרמטר פונקציית next ומחזירה גם היא פונקציה חדשה שמקבלת כפרמטר אוביקט action הקוד (שנראה מעט מסובך):

```
const reduxMiddlewareDemo = store => next => action => {
    // middleware code
};
```

בתוך קוד הmiddleware נוכל לגשת למשתנים הבאים:

1. משתנה ה store ממנו נוכל לקחת את הפונקציות getState ו dispatch
2. משתנה ה action ששלחנו dispatch
3. המשתנה next שהוא פונקציה שמייצגת את המשך שרשרת הmiddlewares.

middleware יכול להסתכל על ה action ולהחליט מה לעשות איתו: בשביל להמשיך טיפול הוא יכול להעביר אותה ל next, בשביל לייצר action חדש הוא יקרא ל dispatch ובשביל להוסיף מידע מהstate הוא יפעיל את getState. אם הmiddleware יחליט שאין להעביר להמשך טיפול, כמו לדוגמא במקרים של ולידציות לא תקינות וכו', הוא יבצע return ריק.

דוגמאות:

middleware שמדפיס את הaction ללוג בכל הודעה.

```
const loggerMiddleware = store => next => action =>
    console.log('ACTION: ', action);
    return next(action);
  };
```

middleware שבודק **לפני** עדכון שם משתמש, ושולח לעדכון רק אם הוא לא ריק. אם הוא ריק, הוא אינו ממשיך טיפול וחוזר באמצעות return.

```
const avoidEmptyUserNameMiddleWare = store => next => action => {
```

```
    if (action.type == 'SET_USERNAME' && action.payload == '')
        return;
    console.log(action.type, ':', action.payload);
    return next(action);
}
```

הוספת middleware לstore

את הmiddlewares שיצרנו, יש לחבר לstore שלנו.

חיבור זה נעשה בעת יצירת הstore, באמצעות הפונקציה (מהספריה redux) applyMiddleware

הפונקציה מקבלת אין-סוף פרמטרים (שהם middlewares) (כמו params)

```
const store = createStore(reducer, applyMiddleware(loggerMiddleware));
```

נקודה חשובה:
סדר הפעלת הmiddlewares, הוא הפוך מסדר שליחתם, כלומר מימין לשמאל.
חשוב לזכור זאת במידה שאחת הפונקציות משנה את הstate, ונרצה להשתמש בstate המעודכן, יש לשלוח את הפונקציה המעדכנת באופן שתקרא לפני, כלומר לשלוח אותה אחרי.

פיתוח single page application באמצעות react-router

מה זה single page application

Single Page Application (תרגום – יישומי דף יחיד) הם יישומי רשת שמטרתם לתת חוויית משתמש יותר מהירה וזורמת ויותר, הדומה יותר לתוכנת מחשב רגילה (שאינה יישום רשת). ביישומים אלו, כל הקוד הדרוש – HTML, CSS, JS – מגיע אל הדפדפן בטעינת דף אחת, ומשאבים נוספים נטענים בצורה דינמית, בדרך כלל כתגובה לפעולות המשתמש. דף האינטרנט לא מבצע טעינה מחדש בשום שלב, אולם כתובת האינטרנט עשויה להשתנות מעט, על מנת לתת למשתמש הבנה יותר טובה של הניווט בדף. יישומים כאלה מציעים חווית משתמש טובה יותר כיוון שמעבר בין עמודים ביישום הוא מהיר יותר בהשוואה לטעינת HTML חדש, אך הם מייצרים מספר אתגרים למפתחים:

באילו בעיות אנו עלולים להיתקל במהלך פיתוח של SPA (single page application)

1. טעינת הדף הראשון עלולה להיות מאד איטית, כיוון שכל תוכן האתר נטען בה.
2. התאמה למנועי חיפוש עשויה להיות מורכבת, כיוון שרק אחרי הפעלת קוד JavaScript אפשר לדעת מה להציג. לא כל מנועי החיפוש יודעים JavaScript ולכן לחלקם יהיה קשה לנווט באתר.
3. ניווט בדפדפן לאחור ושמירת ההיסטוריה – כיון שלא מדובר במעבר עמוד, הדפדפן לא יבצע זאת.
4. זליגות זיכרון ב JavaScript - ביישום רגיל כל מעבר עמוד מאפס את הזיכרון אולם SPA אוביקט הזיכרון יכול לגדול ולגדול לאורך כל חיי היישום, שיכולים להיות מאוד ארוכים.

למעשה, כדי לפתור את (מרבית) הבעיות ניתן להשתמש בספריות מוכנות בהרבה פלטפורמות.

העיקרית בreact היא הספריה react-router

איך עובד react router

ראשית, לפני השימוש בreact router, נתקין את הספריה react-router-dom, באמצעות הפקודה

npm install react-router-dom

ב React Router אנחנו מגדירים אלמנט ראשי מסוג Router שבתוכו יקרה כל הניווט. תפקידו של הראוטר:

1. לזהות מה הדף הנוכחי לפי ה URL
2. לעבד את המידע הזה ולהעביר אותו הלאה לאלמנטים הילדים
3. לטפל במעברי עמודים

React-router צריך להגדיר שני דברים

1. מה יקרה כאשר היישום שלי מגיע לניתוב מסוים (כלומר איזה קומפוננטה צריך להציג עכשיו)
2. אירוע שמעביר את הניתוב ביישום שלי לניתוב מסוים (כדי להציג קומפוננטה מסוימת (סעיף 1)

דוגמא לreact router

```
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,

} from "react-router-dom";

export default function Menu() {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/users">Users</Link>
        </li>
      </ul>
    </nav>

  );
}

export default function Home() {
  return <h2>Home</h2>;
```

```
}

export default function About() {
  return <h2>About</h2>;
}

export default function Users() {
  return <h2>Users</h2>;
}

export default function App() {
  return (
    <Router>
      <div>
        <Menu />
        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/users">
            <Users />
          </Route>
          <Route path="/home">
            <Home />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}
```

למה מיועד הSwitch

כאשר מגדירים את אובייקטי הRoute , יתכנו שני path-ים, שההתחלה שלהם זהה. אם נרצה שבכל פעם יוצג **רק הראשון** מבינהם שמתאים לroute הנוכחי, נשתמש בswitch שיעטוף את כל הRoutes שלי. במידה ולא נשתמש בSwitch, תוצגנה גם קומפוננטות שלא מתאימות בדיוק לניתוב הנוכחי.

מה מכיל הRoute

Route נגדיר כאשר נרצה לומר לreact מה להציג כאשר הוא מגיע לניתוב מסוים

הRoute מכיל attribute בשם path שמכיל את הניתוב (מתחיל ב'/')

הילדים של הRoute (הקומפוננטות שבתוכו) יוצגו כאשר יתקיים הניתוב הנ"ל (שימו לב להערה בסעיף הקודם)

שינוי נתיב באמצעות תגית הLink

ביישום web רגיל, נשתמש לרוב למעבר עמודים באמצעות האלמנט <a href="/..."> אולם אם נשתמש בו בreact-router שלנו, הוא יטען מחדש את העמוד, ולא ישתמש במנגנון של react-router, לכן נשתמש בתגית Link, ואליה נשלח את הניתוב באמצעות הattribute – to. (שוב, מתחיל ב'/')

שינוי נתיב באמצעות קוד

לעיתים, מעבר בין עמודים לא יתרחש כתוצאה ישירה של client event, כמו בתגית הLink, אלא לאחר פעולה מסוימת, כמו לדוגמא, להעביר את היוזר לדף הבית רק לאחר שחזר מהserver response של יוזר תקין.

לצורך כך נשתמש בHigher Order Component שנקרא withRouter.

הפעלת withRouter מוסיפה לprops של הקומפוננטה שלנו אוביקט בשם history.

באמצעות שימוש בפונקציה history.push("path") נבצע route לנתיב רצוי.

לדוגמא:

```
export default withRouter(connect(mapStateToProps)(function Login(props
) {
  const { history } = props;

  function afterValidation(){
    history.push("/home")
  }

}));
```

שינוי נתיב באמצעות Redirect

לעיתים נרצה להוסיף למנגנון הrouting שלנו מעין אבטחה. כלומר אל תכנס לדף מסוים – תחזיר אותי לדף אחר, כל עוד לא מתקיים(ים) התנאי(ם)

לדוגמא: לא לאפשר בתהליך רישום מעבר לעמוד מתקדם יותר, גם במקרה שהוכנס הנתיב הנכון

דוגמא: - כאשר המשתמש מנסה להגיע לדף לוגין, ללא שם משתמש, הוא מוחזר לדף הראשי,

באמצעות return של אלמנט Redirect

```
export default connect(mapStateToProps)(function Login(props) {
  const { username } = props;

  if (!username) {
    return <Redirect to="/" />
  }

  return (
    <div>
      <p>Hello {username}</p>
    </div>
  );
```

```
});
```

ניתן לשלוח בto attribute של הRedirect אובייקט ולא רק ניתוב.

האובייקט יכיל שני מאפיינים:

pathname- הניתוב לעבור אליו

state – מידע נוסף אותו הקומפוננטה אליה מבצעים Redirect תרצה לקבל

דוגמא:

```
export default connect(mapStateToProps)(function Login(props) {
  const { username } = props;

  if (!username) {
    return <Redirect to={{ pathname: '/', state: { error: 'Please enter
 a user name' }}} />
  }

  return (
    <div>
      <p>Hello {username}</p>
    </div>
  );
});
```

את המידע שנשלח, ניתן לשלוף באמצעות props.location, שיחזיר, בנוסף למאפיינים הרגילים גם את הstate. שימו לב שיש לעטוף קומפוננטה זו בwithRouter

ניתן ליצור קומפונטטה שמציגה את השגיאות, לדוגמא:

```
const ShowError = withRouter(function ShowError(props) {
  const { state } = props.location;
  if (state && state.error) {
    return <p>{state.error}</p>
  }
  return false;
});
```

שימוש בפרמטרים בניתוב שלנו

למה משמשים הפרמטרים

לעיתים נרצה לשלוח לrouter, בנוסף לניתוב הרגיל, עוד פרמטים משתנים, לדוגמא, כאשר הניתוב יועבר לדף הצגת מוצר, נרצה לשלוח גם את מזהה המוצר.

לדוגמא:

product/1/

product/7/

product/99/

product/3/

הניתוב בהם זהה, מלבד המזהה, כלומר יוצג עמוד זהה, שבו יוצגו פרטים שונים

איך מגדירים ניתוב עם פרמטרים בRoute

כאשר מגדירים את ה Route, יש לשרשר לו בנוסף לניתוב, גם את הפרמטרים. התו שמבדיל בין "סתם" ניתוב לבין פרמטר, הוא נקודותיים (:) .

לדוגמא

```
<Route path="/product/:id">
    <Product />
</Route>
```

במקרה כזה, הניתוב יהיה Product/anyid

ניתן להוסיף אין-סוף פרמטרים.

הקומפוננטה Product תצטרך לגשת לפרמטר בשם id שהיא תקבל מ React Router ולפי הערך שלו להציג את המוצר המתאים.

איך קומפוננטה ניגשת לפרמטרים של הניתוב

הפונקציה useParams של React Router מאפשרת גישה מהירה לפרמטרים שהגיעו מהנתיב.
הפונקציה מחזירה אוביקט שהמפתח באוביקט הוא שם הפרמטר והערך הוא ערך הפרמטר. אם יש כמה פרמטרים יהיו כמה מפתחות באוביקט. במילים אחרות השורה הבאה מתוך קוד קומפוננטה תתפוס את ה id של המוצר:

```
function Product(props) {
  const { id } = useParams();
}
```