React

TypeScript

מקורות מידע חיצוניים:

- גוגל AI - ג'ימיני
- מכללת האקריו - קורס בניית אתרים

מבוא ותחביר

ריאקט היא טכנולוגית פיתוח בצד הלקוח פותחה ע"י מטא בשנת 2013 (קוד פתוח) ומהשפות המובילות היום בשוק. היא למעשה סיפריית קוד פתוח אפשר לעשות בה שימוש ב-java script . יש לה תכונה מרכזית שונה מJS (בה אנו עובדים עם DOM). בריאקט יש virtual dom שיוצר יעילות בהרצת התוכנית.

כל שינוי שנבצע ב-DOM מוצג אוטומטית בvirtual dom של ריאקט והיא תחליט מתי נכון לבצע עדכון בDOM של הדפדפן על סמך ההגדרות המרכזיות שלה וכל עוד לא הגדרנו אחרת. אנו עובדים בפורמט קבצים מסוג:

java script XML – JSX

type script XML – TSX

ניצור מסמך TSX שבתוכו יהיה HTML,CSS,TS,JS.

React.dev – דקומנטציה

שפת בת react native פיתוח אפליקציות מובייל.

כשאנחנו בונים אתר שמבוסס ריאקט אנחנו עובדים בטכנולוגית SPA:

Single page application .

כל הקוד נטען פעם אחת ובעת מעבר בין קישורים משתנה רק התוכן. זה משפר חווית משתמש ברמות. אם בשפת HTML היינו בונים header ו footer בכל אחד מהעמודים שלנו וכל עמוד היה מקשר לעמוד אחר כאן אנו בונים את כל התוכן פעם אחת ומחליפים רק את בין התכנים הרלוונטים שנרצה להציג למשתמש, הדבר נעשה באמצעות יצירת קומפוננטות – יחידות תוכן.

ברגע שאנחנו מנתבים לדף אחר מה שמשתנה זה רק התוכן למעשה אם אנחנו בדף הבית לוחצים על אודות תופיע קומפוננטה בעלת תוכן שהגדרנו לאודות. אנו יוצרים את כל התוכן מראש ומחליפים בין קומפוננטות ע"י קישורים.

נגדיר כל קומפוננטה באיזה דפים היא תופיע.

יתרונות השפה:

- אתר מהיר מאוד ונגיש למשתמשים

- מאוד קל לתחזוקה – שינויים בתוכן מסוים שחוזר על עצמו בכל הדפים כמו נאב בר, מבצעים פעם אחת וזה רלוונטי לכל הדפים. אגב נעולם הפיתוח הולך לכיוון JS, גוגל נכון להיום יודעת לקרוא JS מה שלא היה בעבר והיא זו שמכתיבה את הטון.
ככל שיש יותר משתמשים על השרת זה יוצר עומס בעבודה עם JS אנו מעבירים את העומס מהשרת אל הדפדפן.
- פחות מידע שמור מהשרת - SPA

הדפדפן לא מכיר את השפה REACT בדומה לtype script נצטרך לקמפל קבצים בעזרת bubble js המרת קובץ ריאקט JSX לקובץ JS המרת קובץ ריאקט TSX לקובץ TS.

שלבים לבניית פרויקט ריאקט:

ראשית נריץ בשורת הפקודות (CMD) של וינדוס על מנת לבדוק שיש לנו גירסאות (version) עדכניות מותקנות במחשב.

node -v

npm -v

במידה ואין לנו גירסא עדכנית – ניתן להוריד מכאן.

- יש לבחור גירסא עדכנית ביותר ותואמת למחשב שלנו.
- בעת פתיחת חלון ההתקנה נבחר NEXT עד הסוף ונתקין בכונן C ברירת המחדל.

כעת נגדיר את ה- vs code שלנו:

- נוריד תוסף בשם simple react snippets
- נלחץ על גלגל שיניים (הגדרות של התוכנה) נבחר settings -> נכתוב emmet <- נרד מעט ל - **Emmet: Include Languages <-** נוסיף אייטם ונציין בו javascript - > נציין javascriptreact ב-value.
יתן לנו השלמה אוטומטית וקיצורים רלוונטים בתוכנה.
- נוריד תוסף לדפדפן react developer tools
יציג לנו את הקומפוננטות והמרכיבים באתר.

שלבים לפתיחת פרויקט ריאקט חדש ב-vs code :

1. נפתח תיקיה חדשה
2. נפתח טרמינל חדש
3. נוודא שאנחנו נמצאים בתוך התיקיה במידה ולא ננווט לתוך התיקיה באמצעות כתיבה CD רווח ואת שם התיקיה. (שמות של תיקיות באנגלית בלבד!)
4. נכתוב שורת קוד בטרמינל
npx create-react-app רווח ושם התיקיה של הפרויקט בה אנחנו נמצאים.
5. כעת נגדיר את type script
--templet type script
נבחר Y (כן להתקין)
6. פקודה להריץ את הפרויקט
npm start

שלבים לפתיחת פרויקט חדש ב- vs code באמצעות VITE :

1. cd to dir
2. npm create vite@latest
3. Choose react and js/ts
4. cd into project
5. npm i
6. npm i bootstrap bootstrap-icons
7. npm run dev

*במידה ונרצה לגשת ל-vsCode דרך שורת הפקודות של ווינדוס, נוודא שאנחנו נמצאים בתיקיה של הפרויקט ונרשום באנגלית את המילה code רווח נקודה.

לשים בתוך קובץ ה-main.jsx את שורות הקוד הבאות לצורך ייבוא של בוטסטראפ לפרויקט:

```
import "bootstrap/dist/css/bootstrap.css;"
import "bootstrap/dist/js/bootstrap.bundle;"
import "bootstrap-icons/font/bootstrap-icons.css;"
```

נסביר קצת על הקבצים שנוצרו לנו בתיקיית הפרויקט:

tsconfig – קובץ הגדרות הקימפול המרת מקבצי JS או TS לקבצי JSX או TSX

Index.html – קובץ פרויקט ראשי

Packed.json – מכיל בתוכו דפנדנסי (עם כל התיקיות שהתקנו)

בקובץ ה-HTML יש "div id="root שמקושר לקבצי TSX שלנו במידה ונוריד אותו כל הפרויקט יעלם. בנוסף הקובץ מכיל script type=module

קובץ הmain.tsx מכיל את כל האינפורטים CSS,App,TSX

Create root מקושר ל "id="root בקובץ ה-HTML

App – זוהי קומפוננטה ראשית, הקובץ מכיל אינפורטים של תיקיות ריאקט, לוגו, קובץ סטייל כל קומפוננטה שניצור אוטומטית תיכנס ל- app.tsx.

Public – תיקיה סטטית ניתן לקשר לתיקיות וקבצים בתוכה ללא צורך לציין את שמה בכתובת הניתוב – URL.

כל קומפוננטה היא למעשה פונקציה.

בריאקט יש קובץ אחד שעליו יש HTML,CSS,TS.

כשאנחנו רוצים לכתוב בשפות השונות החוקים יהיו:

- **Html – בתוך סוגרים עגולים ( )**
- **JS – בתוך סוגריים מסולסלים { }**
- **TS – בתוך משולשים בדומה לתגית < >**

כל מה שמעל ה-return יהיה שפת JS

בתוך ה-return למעשה כל מה שמתחת ישולב במספר שפות HTML.JS,TS.

בתוך הreturn שהקומפוננטה מחזירה יכולה להיות רק תגית אב אחת נוכל להשתמש בתגית div מסוג קונטיינר לצורך הדוגמא או בפרגמנט fragments בדומה לתגית HTML ריקה פותחת וסוגרת ובתוכה נכניס את כל הקוד שלנו.

דוגמא לפרגמנטס:

```
return (
  <>
    </>
);
```

שגיאות בקוד – מקריסות את התוכנית

WARNING – מהווה אזהרה כמו למשל משתנים שהצהרנו עליהם ולא השתמשנו בהם בקוד.

ניצור תיקיית קומפוננטות (components) בתוך תיקיית ה-SRC

לאחר מכן ניצור קובץ חדש TSX

קיצור במסמך TSX הקלדה של המקשים הבאים: FC+TAB - תיפתח לנו קומפוננטה חדשה.

הסמן יהבהב ב-3 מקומות נוכל מיד להקליד את שם הקומפוננטה והיא תיכתב במקומות אלו.

אנו כותבים את שם הקומפוננטה בדיוק באותו השם הקראנו לקובץ

נוכל לעמוד על הקו האדום המסולסל כדי לתקן שגיאה מסוג ייבוא אימפורט נלחץ על:

1. Quick fix
2. add import from

ואוטמטית נקבל את ה-import המתאים למסמך והקומפוננטה.

בריאקט יש התייחסות ל-function components

Interface – מילה שמורה של TS המגדירה מבנה חוקיות מסוימת בקוד.

Props – מאפיינים של קומפוננטה קיצור של property פרמטר שליחה או קבלה.

קומפוננטה ראשית יכולה לקרוא לקומפוננטות אחרות.

<u>כיצד לייבא קומפוננטה ל-App ?</u>

- נכתוב את שם הקומפוננטה בדיוק כפי שהיא רשומה
- קומפוננטה חייבת אות ראשית גדולה
- בקומפוננטה התגית הפותחת היא גם התגית הסוגרת למשל <navbar/>

במידה ונרצה לפתוח שורת טרמינל חדשה נלחץ על כפתור ה+ ולאחר מכן new terminal או בקיצור מקשי מקלדת : Ctrl+Shift+Backtick (בקטיק ממוקם - מצד שמאל של הסיפרה 1 ומעל המקש TAB)

Props תמיד נקבל כאובייקט בתוך משולשים למשל <props> - זהו סימן לאובייקט טייפ סקריפט.

נוכל לבצע distractering לפירוק האוביקט או המערך:

- באובייקט דיסטרקצ'רינג חשוב השמות של המפתחות
- במערך דיבטרקצ'רינג חשוב הסדר של האינדקסים

שימו לב שורת קוד HTML עם הפונקציה ()onClick תפעיל מיד את הפונקציה בעת רינדור של ריאקט ולא בעת לחיצה, לכן נוריד את הסוגריים כדי לאפשר הפעלת הפונקציה רק בעת לחיצה.

Inline CSS – נרשום את המילה סטייל שווה ונפתח סוגריים מסולסלים פעמיים style={{}} פעם ראשונה מכיוון שאנחנו רוצים לכתוב JS ופעם שניה מכיוון שאנחנו כותבים אובייקט, בתוך הסוגרים המסולסלים נוכל להגדיר את המאפיינים וערכים של הסטייל.

Class – היא מילה שמורה ב-JS לכן נשתמש לכן כאשר נרצה להגדיר קלאס ב-HTML נשתמש ב- className

For – היא מילה שמורה ב-JS ללולאות לכן כאשר נרצה להגדיר יעד בHTML נשתמש ב- htmlFor

קומפוננטה

החלק מעל ה- return ישמש אותה להגדרת משתנים ופוקציות זהו החלק הלוגי..

החלק שמתחת ל-return ישמש אותנו להצגת הקומפוננטה זהו החלק של התצוגה.

במידה ונרצה לקרוא לפונקציה **onClick** שמקבלת פרמטר ובכדי לא להפעיל מיידית את הפונקציה בעת ריצת הקוד ניצור בתוך סוגריים מסולסלים פונקציית callback למעשה פונקציה onclick תקרא לפונקציה אחרת עם הפרמטר שאנחנו רוצים להעביר.

```
<button
        className="btn btn-warning"
        onClick={() => {
          sayHolla("Maor");
        }}>
        Say Hello
      </button>
```

href – יוצר רענון של הדף לכן בריקאט אנחנו נשתמש בראוטינג נלמד על כך בהמשך.

קיצור מקשי מקלדת fc – יצירת קומפונטה למי שעובד עם TSX

קיצור מקשי מקלדת ffc – יצירת קומפוננטה למי שעובד עם JSX

onChange – ברגע שהשדה קלט משתנה אנחנו רוצים לעשות משהו עם המידע, כל שינוי ב- onChange יוצר לנו אובייקט שמכיל את האירוע בתוך האובייקט יש מפתח target שבתוכו נמצא הvalue

return – עוצר את העיבוד של התוכנית, אנחנו יכולים להגדיר לפני הריטרן של הקומפוננטה שאם תנאי מסוים מתקיים (מחזיר true) אז תחזיר return כל מה שנרצה. במידה והתנאי לא יתקיים הוא אוטומטית ימשיך ל-return של הקומפוננטה וכך אנחנו מגדירים מה יוצג בדף באיזה תנאי.

לדוגמא:

אם isAdmin מחזיר true אז תציג Hello Admin.

אם isAdmin מחזיר false אז תציג Hello User.

פעולה זו נקראת בריאקט **conditional rendring – עיבוד מותנה (התנאי יקבע איזה תוכן יוצג)**

```
const Welcome1: FunctionComponent<Welcome1Props> = ({ isAdmin }) => {
  if (isAdmin) {
    return (
      <>
        <h4>Hello Admin</h4>
        <button className="btn btn-primary">Admin Panel</button>
      </>
    );
  }

  return (
    <>
      <h4>Hello User</h4>
      <button className="btn btn-warning">Products Catalog</button>
    </>
  );
};
```

and operator – נשתמש בו כשיש תנאי אחד שצריך להתקיים. (&&)

ב-JS אנחנו יודעים על האופרטור אנד שבמידה והתנאי הראשון מתקיים הוא ממשיך לתנאי הבא ובמידה והשני מתקיים הוא ממשיך לבדוק את הבא וכן הלאה. בריאקט במידה ותנאי מתקיים

למשל אם isAdmin מחזיר true תציג hello massage from welcome 2

```
const Welcome2: FunctionComponent<Welcome2Props> = ({ isAdmin }) => {
  return (
    <>
      {isAdmin && (
```

```
        <>
          <h4>Hello massage from welcome 2</h4>
          <button className="btn btn-danger">Go to admin panel</button>
        </>
      )}
    </>
  );
};
```

האופרטור הלוגי && בריאקט משמש ככלי רב עוצמה לשליטה בתנאי על מנת להציג או להסתיר תוכן. הוא מאפשר לנו ליצור מבנים מותנים באופן קומפקטי וברור, מה שהופך את הקוד שלנו לקריא יותר וקל יותר לתחזוקה.

כיצד זה עובד?

בגדול, האופרטור && עובד בצורה הבאה:

אם הביטוי משמאל לאופרטור הוא true: הביטוי מימין לאופרטור יוחזר.

אם הביטוי משמאל לאופרטור הוא false: הביטוי משמאל יוחזר (ששווה ל-false).

טרנרי Ternary – במידה ויש תנאי שיתקיים או לא נשתמש בטרנרי ובכך נשלוט מה יוצג בדפדפן.

```
const hasError = false;
const message = hasError ? <p>אירעה שגיאה</p> : <p>הפעולה בוצעה בהצלחה</p>;

return <div>{message}</div>;
```

map – בעזרת המתודה מאפ נוכל לעשות מניפולציה על מערכים ולהציג את הערכים שלהם ב-HTML. למשל במידה ויש לנו מערך של אובייקטים מסוג users נוכל לבצע מניפולציה על המערך באמצעות map שתרצו על כל איברי המערך ותחזיר פיסקה המכילה את user.name את המפתח שם של אותו יוזר ספציפי, כשתסתיים הריצה על המערך נקבל פסקאות עם כל השמות של היוזרים.

Render - רנדר

unique key – רענון של התוכן ספציפי בדפדפן לפי מפתח יחודי מסוים.

במידה ולא נשתמש בkey ריאקט תרענן את כל הדף.

מהו מפתח ייחודי?

מפתח ייחודי (unique key) הוא תכונה מיוחדת שמוסיפים לאלמנטים ברשימה בריאקט. הוא משמש את ריאקט כדי לעקוב אחר כל אלמנט בנפרד ולנהל את ה-DOM בצורה יעילה. המפתח עוזר לריאקט להבין אילו אלמנטים נוספו, הוסרו או שונו ברשימה, וכך הוא יכול לבצע את השינויים המינימליים ביותר ב-DOM.

חשוב: המפתח חייב להיות ייחודי לכל אלמנט ברשימה. כלומר, לא יכולים להיות שני אלמנטים עם אותו מפתח.

מדוע מפתחות ייחודיים חשובים?

יעילות: מפתחות מאפשרים לריאקט לבצע אופטימיזציות ולעדכן רק את האלמנטים שבאמת השתנו, במקום לעבור על כל הרשימה מחדש.

סדר: מפתחות עוזרים לריאקט לשמור על הסדר הנכון של האלמנטים ברשימה, גם כאשר הרשימה משתנה.

מניעת באגים: ללא מפתחות, ריאקט עלולה לטעות בזיהוי האלמנטים וליצור באגים בלתי צפויים.

<u>רינדור רשימות עם ומבלי מפתחות</u>

ללא מפתחות:

כאשר מראנדרים רשימה ללא מפתחות, ריאקט מתייחס לכל האלמנטים כאל אלמנטים חדשים בכל רנדור. זה יכול להוביל לביצועים גרועים ולבעיות בניהול המצב של הרכיבים.

```
const numbers = [1, 2, 3];

const listItems = numbers.map((number) => <li>{number}</li>);

return <ul>{listItems}</ul>;
```

עם מפתחות:

כאשר מוסיפים מפתח ייחודי לכל אלמנט ברשימה, ריאקט יכולה לעקוב אחר כל אלמנט בנפרד ולנהל את השינויים בצורה יעילה.

```
const numbers = [1, 2, 3];

const listItems = numbers.map((number) => <li
key={number}>{number}</li>);
```

בחירת מפתח:

הבחירה במפתח תלויה בנתונים שלך. עדיף לבחור מפתח שייחודי לכל אלמנט ושהוא לא ישתנה במהלך חיי האלמנט. דוגמאות למפתחות טובים:

ID: אם לכל אלמנט יש ID ייחודי.

אינדקס: רק אם האינדקס של האלמנט לא ישתנה.

שילוב של מספר שדות: אם אין שדה יחיד ייחודי, ניתן לשלב כמה שדות כדי ליצור מפתח ייחודי.

כלל חשוב מאוד!

כאשר אנחנו רוצים להחזיר מפונקציה HTML היא תכתב בצורה הבאה ()<=() נפתח סוגריים עגולים ולא מסולסלים מכיוון שהפונקציה מחזירה HTML.

דוגמא:

```
return (
    <div id="container-dishes">
      {dishes.map((dish) => (
        <div className="card" style={{ width: "18rem" }}>
          <img
            className="card-img-top"
            src="https://raameatshop.co.il/wp-
content/uploads/2021/08/WhatsApp-Image-2023-11-28-at-13.59.08.jpeg"
            alt="Card image cap"
          />
          <div className="card-body">
            <h5 className="card-title">{dish.foodName}</h5>
            <p className="card-text">{dish.price}</p>

            <a href="#" className="btn btn-primary"
onClick={youClicked}>
              Click Me!
            </a>
          </div>
        </div>
      ))}
    </div>
  );
```

דבר נוסף בעת שאנחנו עושים השמה למשתנה ריאקט לא תבצע רינדור ולכן הערך החדש לא יוצג בדפדפן לשם כך נצטרך להשתמש בהוקס.

HOOKS

רינדור – עיבוד של הקוד מחדש, כל שינוי של state יגרור רינדור של הקומפוננטה.

Hooks – פונקציות עזר שאנחנו יכולים להשתמש בהם כדי להשיג מטרות מסוימות.

useState – תאפשר לנו לתת משתנה כ-state וכאשר יתעדכן useState תבצע רינדור מחדש לאפליקצי למעשה לקומפוננטה ותהפוך אותם להיות דינמיים.

מבצע רינדור לכל הקומפוננטה, ברגע שstate משתנה הקומפוננטה עוברת רינדור באופן יעיל ורק מה שצריך להתעדכן משתנה.

במבנה של useState אנחנו נצטרך שם משתנה, ושם הפונקציה שתעדכן את המשתנה ניתן גם להוסיף ערך התחלתי.

תפקידה של הפונקציה לבצע השמה של אותו State .

סוגריים ריקים – אין ערך התחלתי

סוגריים המכילות ערך – מבצע השמה של ערך התחלתי למשתנה.

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

בדוגמא הזו למעשה נתנו ערך התחלתי מחרוזת "maor" למשתנה name .

בעת הפעלת הפונקציה setName הערך ההתחלתי ישתנה.

כל שינוי של סטייס מבצע רינדור מחדש.

useEffect – קטע קוד שירוץ בכל פעם שיש רינדור, מבצע פעולה כתוצאה מרינדור של הקומפוננטה.

אחרי הסוגריים המסולסלים של ההוק ובתוך הסוגריים של יוז אפקט, נפתח סוגריים מרובעות [] שהם למעשה דפנדנסי במידה והסוגריים ריקות היוז אפקט יפעל רק בעת הרינדור הראשוני, במידה ובסוגריים קיימים שמות משתנים היוז אפקט יפעל בכל פעם שיש רינדור מחדש של המשתנה. ניתן להפריד בין שמות משתנים ע"י פסיקים כמו במערך.

לצורך מעקב על רנדרים יש צורך בשמות משתנים של State.

רינדור ראשוני – בעת שהקומפוננטה נטענת בפעם הראשונה.

מבנה – פותחים סוגריים של יוז אפקט ובתוכו פונקציית חץ arrow function.

DOM – זהו עץ תגיות בJS שמשפיע על ה-HTML.

נראה כעת דוגמא ליוז אפקט.

```
useEffect(()=>{
    console.log(name)
  },[name]);
```

בכל פעם שהסטייס name ישתנה יודפס בקונסול name.

Custom hook – יצירת הוקר מותאם אישית משלנו.

יוצרים תיקיה בתוך ה-SRC בשם hooks.

נפתח קובץ חדש למשל: useFetch.tsx

ניצור משתנה ששווה למבנה של פונקציית חץ.

שם המשתנה חייב להיות באותו השם של קובץ ה-custom hook שלנו.

כל רינדור של סטייס.

ההוק יכול יפעל במצבים הבאים וכפי שנגדיר:

1. Use effect בלי דפנדנסי יפעל בכל רינדור של הקומפוננטה

2. use effect עם דפנדנסי של מערך ריק יפעל ברינדור הראשוני בלבד
3. use effect עם דפנדנסי של משתנים מסוימים יפעל בעת רינדור של state מסוים.

Dead lock – לופ אינסופי של רנדרים שיכול גם להקריס את התוכנית.

Routing

כפי שאנחנו מכירים ב-HTML יצירת קישור ע"י href במעבר בין דפים גוררת טעינה מחדש של הדף.

השימוש בראוטינג בריאקט יעזור לנו לבנות one single page ושלוט בתוכן שיוצג בכל פעם כשיש לחיצה על קישור מסוים. זאת אומרת אנו נבנה את כל התוכן של האתר בדף אחר ונציג את הקומפונטטות המכילות תוכן רלוונטי בעת לחיצה על קישור מסוים.

צעדים להתקנה:

1. פתיחת טרמינל חדש
2. Npm I react-router-dom
3. נוסיף אימפורטים ב-app

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
```

React Router DOM היא ספרייה פופולרית ביותר המשמשת לניהול ניווט ביישומי React. היא מאפשרת לנו ליצור יישומים בעמוד אחד (Single-Page Applications, SPA) שמרגישים כמו אתרים מרובי דפים, מבלי לטעון מחדש את כל העמוד בכל פעם שהמשתמש עובר בין מסכים.

מה זה ראוטינג?

ראוטינג, במונחים של פיתוח Web, מתייחס לתהליך של התאמת כתובות URL לדפים או רכיבים שונים ביישום. כשאתה גולש באתר, הכתובת בשורת הכתובת של הדפדפן משקפת את המיקום הנוכחי שלך באתר. React Router DOM מאפשר לנו ליצור התאמה כזו ביישומים שלנו.

למה React Router DOM?

ניווט חלק: מאפשר מעבר בין מסכים ללא טעינה מחדש של כל העמוד, מה שמוביל לחוויית משתמש חלקה יותר.

מבנה ברור: עוזר לארגן את היישום למודולים שונים, כל אחד עם ה-URL שלו.

ניהול מצבים: מאפשר לשמור ולנהל מצבים שונים של האפליקציה בהתאם ל-URL הנוכחי.

תמיכה במסלולים דינמיים: מאפשר יצירת מסלולים דינמיים, כמו למשל /products/:id, כדי להציג מידע ספציפי למשתמש.

רכיבים עיקריים ב-React Router DOM:

BrowserRouter: הרכיב העליון שמקיף את כל האפליקציה ומאפשר ניווט רגיל בדפדפן.

Routes: מכיל את כל המסלולים האפשריים לניווט באפליקציה. (קונטיינר של כל הראוטים)

Route: מגדיר מסלול ספציפי לניווט ואיזה רכיב צריך להציג כאשר המשתמש נמצא במסלול זה.

Link: יוצר קישורים פנימיים בתוך האפליקציה, ומעדכן את ה-URL ללא טעינה מחדש של העמוד. (מחליף את תגית a href)

*יש לבצע אימפורט של Link מתוך ריאקט ראוטר דום:

```
import { Link } from "react-router-dom";
```

NavLink : רכיב מיוחד ב-React Router DOM שמשמש ליצירת קישורים פנימיים ביישומים שלך. הוא מבוסס על הרכיב הבסיסי Link, אך מוסיף לו פונקציונליות נוספת: הצגת מצב פעיל. (מחליף את תגית nav)

***יש לבצע אימפורט של NavLink מתוך ריאקט ראוטר דום:**

```
import { NavLink } from "react-router-dom";
```

***NavLink חייב להיות בתוך רכיב browser router ראשי.**

מה הכוונה במצב פעיל?

כאשר משתמש לוחץ על קישור ב-NavLink, הקישור הזה מסומן כ"פעיל". סימון זה מאפשר לך להוסיף סגנונות עיצוב ייחודיים לקישור הפעיל, כדי שהמשתמש יבין איזה חלק מהאתר הוא נמצא בו כרגע. לדוגמה, אתה יכול לשנות את צבע הטקסט, להוסיף רקע צבעוני, או לשנות את הגופן.

למה להשתמש ב-NavLink?

חוויית משתמש משופרת: סימון הקישור הפעיל מספק למשתמש משוב חזותי ברור על מיקומו הנוכחי באפליקציה.

קוד נקי ומאורגן: NavLink מפשט את תהליך הוספת סגנונות לקישורים פעילים, ומאפשר לך לשמור על קוד ה-CSS שלך מסודר.

גמישות: NavLink מציע מספר תכונות להתאמה אישית של המראה וההתנהגות של הקישורים הפעילים.

```
import { NavLink } from "react-router-dom";

function MyNavbar() {
  return (
    <nav>
      <NavLink to="/">Home</NavLink>
```

```
      <NavLink to="/about">About</NavLink>
      <NavLink to="/contact">Contact</NavLink>
    </nav>
  );
}
```

to: תכונה זו מציינת לאיזה מסלול הקישור יוביל.

activeClassName: תכונה זו מאפשרת לך להוסיף שם מחלקה לקישור כאשר הוא פעיל. אתה יכול להשתמש במחלקה הזו כדי להגדיר את הסגנונות הרצויים ב-CSS.

activeStyle: תכונה זו מאפשרת לך להוסיף סגנונות אינליין לקישור כאשר הוא פעיל.

תכונות נוספות ב-NavLink:

exact: מאפשר לך להגדיר אם הקישור יהיה פעיל רק כאשר ה-URL תואם בדיוק למסלול.

strict: מאפשר לך לקבוע אם יש להתחשב בסלש בסוף ה-URL כאשר בודקים התאמה.

style: מאפשר לך להוסיף סגנונות אינליין לקישור.

className: מאפשר לך להוסיף מחלקה לקישור בכל מצב.

```
<NavLink to="/" activeClassName="active">
  Home
</NavLink>;
```

<u>כעת נראה דוגמא למבנה של browser router:</u>

```
import React from "react";
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";

function Home() {
  return <h2>Home Page</h2>;
}

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

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
      </nav>
      <Routes>
```

```
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

הסבר:

- BrowserRouter עוטף את כל האפליקציה.
- nav מכיל את הקישורים ל-Home ו-About.
- Routes מכיל את כל המסלולים האפשריים.
- Route מגדיר את המסלול ואת הרכיב המתאים.
- Link יוצר קישורים פנימיים.

***ניתן להגדיר ל-route ניווט ליותר מקומפוננטה אחת ע"י הוספת תגית אב או פרגמנט המכילה את כל הקומפוננטות הרצויות.**

```
<Routes>
        <Route
          path="/"
          element={
            <>
              <Home />
              <About />
            </>
          }
        />
      </Routes>
```

על מנת שגולש לא יכניס בכתובת URL כל כתובת שירצה ויקבל את העמוד הראשי נוסיף קומפוננטה חדשה NotFound בסוף של routes כדי לאפשר מצב של מעבר על כל הראוטים ובמידה ולא נמצא נציג דף לא נמצא, נוכל לעצב אותו כדי שנרצה.

```
<Route path="*" element={NotFound}/>
```

כך תיראה למשל הקומפוננטה:

```
import { FunctionComponent } from "react";

interface NotFoundProps {}

const NotFound: FunctionComponent<NotFoundProps> = () => {
  return <h2 className="display-2">Page Not Found - 404</h2>;
```

```
};

export default NotFound;
```

ניתן להוסיף פרמטר ל-route ע"י האופרטור נקודותיים (:/) לכתובת URL מסוימת:

```
<Route path="/topseries/:name" element={<SeriesDetails />} />
```

לאחר ששלחנו פרמטר יש להיכנס לקובץ TSX שמקבל את הפרמטר ולהשתמש בו בעזרת useParams ומכיוון שאנחנו מקבלים אובייקט נבצע אובג'קט דיסטרקצ'רינג לפרמטר ונשמור אותו בתוך משתנה כדי שנוכל להשתמש בו ולהשוות אותו לערכים אחרים.

```
const { name } = useParams();
  let obj = series.find((object) => {
    return object.name === name; // החזר את תוצאת ההשוואה
  });
```

הדוגמא הזו שמרנו את הפרמטר name בתוך משתנה ובאמצעות המתודה find מצאנו את האובייקט במערך שהשם שלו שווה לשם שהתקבל כפרמטר.

וכאן למעשה בקומפוננטה שמכילה את הלינק שלחנו ארגיומנט לכתובת ה-URL:

```
<Link to={`/topseries/${oneSeries.name}`}>{oneSeries.name}</Link>
```

זאת אומרת השלבים:

1. שולחים ארגיומנט חדש לכתובת ה-URL כמו למשל name
2. נכנסים לניתוב ה-route ב-app ומגדירים path לאותה כתובת URL בתוספת פרמטר ונרצה להשתמש בו
3. הולכים לקומפוננטה שבה אנו רוצים להשתמש בפרמטר וע"י שימוש בuseParams שומרים את הערך בתוך משתנה, כעת נוכל להשתמש בו לצורך השוואות.

נסטד ראוטינג – נתיבים מכוננים (נתיב בתוך נתיב).

ראוט ראשי – Route index

נתיב מכונן – Route path (פנימי)

Path – נתיב שבא אחרי הדומיין

Browser router – מאזין לURL

Routes – מפנה ל-route המתאים

Route – הנתיב אליו פונים

```
<Route path="/series">
```

```
          <Route index element={<TopSeries />} />
          <Route path=":name" element={<SeriesDetails />} />
      </Route>
```

שימוש בnavigate לניווט:

- פרמטר (-1) יחזיר צעד אחד אחורה
- פרמטר (/) יקח אותנו לדף הבית ראשי

```
const NotFound: FunctionComponent<NotFoundProps> = () => {
  let navigate = useNavigate();
  return (
    <>
      <h2 className="display-2">Page Not Found - 404</h2>
      <button className="btn btn-primary" onClick={() => navigate(-1)}>
        Go back
      </button>
      <button className="btn btn-warning" onClick={() =>
navigate("/")}>
        Go Home
      </button>
    </>
  );
};
```

התקנת מערכת axios

npm i axios

בתוך תיקית הפרויקט, תיקיית service

נגדיר קובץ שירות service.tsx שבוא כל הקריאות לשרת בקובץ אחד, ובתוכו ניצור export לפונקציות get,set,update,delete וכו' שמחזירות axios ל-api.

דוגמא לפונקצית get שמחזירה אקסיוס:

```
import axios from "axios";

const api = "https://jsonplaceholder.typicode.com/users/";

// get all users from api
export function getAllUsers() {
  return axios.get(api);
}

// get specific user by id
export function getUserById(id: number) {
  return axios.get(`${api}/${id}`);
```

לאחר שהגדרנו קובץ servis ניתן לקרוא לפונקציות בקבצים השונים:

```
import { getAllUsers,getUserById } from "../services/userService";
```

```
useEffect(() => {
      getUserById(id)
        .then((res) => setUser(res.data))
        .catch((err) => console.log(err));

      getAllUsers()
        .then((res) => setUsers(res.data))
        .catch((err) => console.log(err));
    }, []);
```

קומפוננטה register (טופס הרשמה)

- נוכל להגדיר בשדה input (שהיא תגית פותחת וגם סוגרת) אפשרות להשלמה אוטומטית של פרטים תיהיה פעילה או כבויה:

```
<input type="email" autoComplete="on" />
<input type="email" autoComplete="off" />
```

- סיפריות:
  - Formik – טפסים עם ריאקט (הגדרת הטופס)
  - yup – ולידציה של טפסים (הגדרת השדות)
- כפתור submit – שליחה
- ניצור 2 סטייטים (state) אימייל וסיסמא.
  - Email
  - Password
- onChange – יפעל את set בכל אירוע שבו מתבצע שינוי ובכך יהיה לנו את הערך העדכני בתוך המשתנה.
- onSubmit – פונקציה חדשה שתפעל בעת לחיצה על כפתור שליחה hendel register
- אובייקט של האירוע hendelRegister נקרא ב-typeScript : formEvent
- נבטל את טעינת הדף הדיפולטיבית בעת לחיצה על כפתור שליחה e.preventDefalt

דוגמא לקומפוננטה register:

```
import { FormEvent, FunctionComponent, useState } from "react";

interface RegisterProps {}

const Register: FunctionComponent<RegisterProps> = () => {
  let [email, setEmail] = useState("");
  let [password, setPassword] = useState("");
```

```tsx
  let handleRegister = (e: FormEvent) => {
    e.preventDefault();
    console.log({ email, password });
  };
  return (
    <div className="container w-50">
      <h2 className="display-4">Register</h2>
      <form className="text-center" onSubmit={handleRegister}>
        <div className="form-floating mb-3">
          <input
            type="email"
            className="form-control"
            autoComplete="off"
            id="email"
            placeholder="name@example.com"
            onChange={(e) => setEmail(e.target.value)}
          />
          <label htmlFor="email">Email address</label>
        </div>
        <div className="form-floating mb-3">
          <input
            type="password"
            className="form-control"
            id="password"
            placeholder="password"
            onChange={(e) => setPassword(e.target.value)}
          />
          <label htmlFor="password">Password</label>
        </div>

        <button className="btn btn-success" type="submit">
          Register
        </button>
      </form>
    </div>
  );
};

export default Register;import { FormEvent, FunctionComponent, useState
} from "react";

interface RegisterProps {}

const Register: FunctionComponent<RegisterProps> = () => {
  let [email, setEmail] = useState("");
  let [password, setPassword] = useState("");
  let handleRegister = (e: FormEvent) => {
    e.preventDefault();
    console.log({ email, password });
  };
  return (
```

```tsx
    <div className="container w-50">
      <h2 className="display-4">Register</h2>
      <form className="text-center" onSubmit={handleRegister}>
        <div className="form-floating mb-3">
          <input
            type="email"
            className="form-control"
            autoComplete="off"
            id="email"
            placeholder="name@example.com"
            onChange={(e) => setEmail(e.target.value)}
          />
          <label htmlFor="email">Email address</label>
        </div>
        <div className="form-floating mb-3">
          <input
            type="password"
            className="form-control"
            id="password"
            placeholder="password"
            onChange={(e) => setPassword(e.target.value)}
          />
          <label htmlFor="password">Password</label>
        </div>

        <button className="btn btn-success" type="submit">
          Register
        </button>
      </form>
    </div>
  );
};

export default Register;
```

formik & yup

התקנה:

1. npm i formik yup
2. import formik from "formik"
3. import * as you from "yup" – ייבוא של כל המתודות

yup – ולידציה של שדות בטפסים

formik – הגדרה של טפסים

ניצור קומפוננטה חדשה Login

בקובץ app.tsx נגדיר ראוט חדש

```
<Route path="/login" element={<Login />} />
```

נוסיף בקומפונטת Login לינק לקומפונטת register מחוץ לטופס שיצרנו.

```
<span>
New user? Please <Link to="/register">register</Link> first.
</span>
```

נגדיר לוגיקה מאחורי הטופס באמצעות formic (מעל ה-return)

- ניצור משתנה בשם formic שיש לו type מסוג fornikValues
- נייבא את האימפורטים הרלוונטים formik ו – yup

formik צריך לקבל 3 אובייקטים בתוך האובייקט שלו:

InitialValues : ערך התחלתי לשדות ניתן להגדיר ערכים ריקים.

ValidationSchema : ולידיציה של כל אחד מהשדות.

onSubmit : הגדרת הפעולות שיבוצעו בעת לחיצה על כפתור submit

Formik יודע לזהות את השדות בטופס לפי אטריביוט מסוג name.

התחלת עבודה עם yup

בתוך validationSchema ובתוך סוגריים מסולסלים נגדיר באילו מתודות אנחנו רוצים להתשמש.

לדוגמא:

Required – שדה חובה

Default – ערך דיפולטיבי

Positive - חיובי

Email,url,date כללי ולידציה לשדות

Mix – ניתן לחבר מספר כללים

import}

mixed,

string,

number,

boolean,

bool,

date,

object,

array,

ref,

lazy,

{from 'yup;'

onSubmit הפונקציה מקבלת ערכים מחזירה פעולות שנגדיר.

Formik – אובייקט שבתוכו מכיל מתודות כמו:

Dirty – קוד מלוכלך (הקלדה ויציאה מהשדה)

Blur – אירוע בטופס (לחיצה על שדה ויציאה ממנו)

isValid – תקין

Change – אירוע שינוי בטופס

Reset – איפוס של השדות

Values – אובייקט שמכיל את המפתחות והערכים של הטופס לדוגמא אימייל וסיסמא

כל עוד לא נגענו בטופס לא יהיו לנו שגיאות, לאחר שניגע בטופס המתודות של formik יפעלו ויציגו שגיאות בהתאם.

מתחת ל-label נוסיף פסקה עם קלאס text-danger

Regex generator – כלי AI שמייצר לנו כללים ל-match

בעת לחיצה על submit נבטל רענון של הדף ונאפס את שדות הטופס.

נבטל את הפעילות של כפתור שליחה בזמן שהטופס לא תקין או מלא.

```
import { FunctionComponent } from "react";
import { Link } from "react-router-dom";
import { FormikValues, useFormik } from "formik";
import * as yup from "yup";

interface LoginProps {}

const Login: FunctionComponent<LoginProps> = () => {
  const formik: FormikValues = useFormik<FormikValues>({
    initialValues: {
      email: "",
      password: "",
    },
```

```
  validationSchema: yup.object({
    email: yup.string().required().email(),
    password: yup
      .string()
      .required()
      .min(9, "Password too short! shuld be at least 9 charcters")
      .matches(
        /^(?=.*[!@#$%^&*(),.?":{}|<>]).*$/,
        "Password must contain at least one special charecter"
      ),
  }),
  onSubmit: (values, { resetForm }) => {
    console.log(values);
    // formik.resetForm();
    resetForm();
  },
});

return (
  <>
    <div className="container w-50 border p-4 my-5">
      <h4 className="display-4">Login</h4>
      <form className="mb-4" onSubmit={formik.handleSubmit}>
        <div className="form-floating mb-3">
          <input
            type="email"
            name="email"
            className="form-control"
            id="email"
            placeholder="name@example.com"
            autoComplete="on"
            onChange={formik.handleChange}
            onBlur={formik.handleBlur}
            value={formik.values.email}
          />
          <label htmlFor="email">Email address</label>
          {formik.touched.email && formik.errors.email && (
            <p className="text-danger">{formik.errors.email}</p>
          )}
        </div>
        <div className="form-floating mb-3">
          <input
            type="password"
            name="password"
            className="form-control"
            id="password"
            placeholder="Password"
            autoComplete="off"
            onChange={formik.handleChange}
            onBlur={formik.handleBlur}
            value={formik.values.password}
```

```
            />
            <label htmlFor="password">Password</label>
            {formik.touched.password && formik.errors.password && (
              <p className="text-danger">{formik.errors.password}</p>
            )}
          </div>
          <button
            disabled={!formik.dirty || !formik.isValid}
            type="submit"
            className="btn btn-success"
          >
            Login
          </button>
        </form>
        <span>
          New user? Please <Link to="/register">register</Link> first.
        </span>
      </div>
    </>
  );
};

export default Login;
```

בformik באופן אוטומטי הדפדפן לא מבצע טעינה מחדש בעת שליחה של הטופס.

ב-onSubmit אנחנו נשתמש ב-resetForm כדי לאפס את שדות הטופס לאחר לחיצה על כפתור שליחה.

כאשר אנחנו רוצים לעדכן בפורמיק פרטים שמגיעים מהשרת נשתמש במתודה:

enableReinitialize: true

הערכים יעודכנו והטופס יטען מחדש לאחר קבלת הפרטים מהשרת.

הוצאת API לקובץ סביבה חיצוני:

- ניצור קובץ חדש בתיקיה הראשית בשם env.
- בתוך הקובץ ניצור משתנה סביבה לדוגמא VITE_API
- ניגש לקובץ service שלנו ונגדיר משתנה חדש:
  - Const api:string = improrт.meta.env.VITE_API

הגדרת קובץ אחד עם משתנה לכל לקובץ לא תיהיה גישה לאף אחד בשרת הוא יהיה מאובטח ולא נעלה אותו בהמשך נדבר על הקבצים האלה באמת אבטחה וקבצים מוצפנים.

ידע כללי:

Token – זהו מידע מוצפן

Path – מה שמגיע אחרי הדומיין

Router – מאזין ל-URL

בעבודה עם **json-server** נרשום בטרמינל שורת קוד שיסונכרן עם הקובץ JSON שלנו נציין גם את השם שלנו לדוגמא שם הקובץ db.json הפקודה תראה כך:

Json-server –watch db.json

משתנה דגל – עבר שינוי או לא עבר שינוי true/false

useState חדש CHANGE – בודק אם היה שינוי

Export function – על מנת לייצא פונקציה מחוץ לקובץ

כשאנחנו פונים החוצה מהתוכנית למשל קריאה לAPI נשתמש בפונקציה אסינכורנית שתפעל כאשר נקבל בחזרה מהשרת אובייקט של הבקשה הצלח/כשלון .

React tostify

סיפרייה שנועדה לשיפור חווית המשתמש ומתן מחוות לגולש.

התקנת ריאקט tostify:

npm i react-tostify

יש לנו 2 דברים עיקריים בטוסטיפיי:

1. טוסט קונטיינר – ייבוא קומפוננטה עם פרופס, זה למעשה הHTML התצוגה עצמה של ההודעה.
2. טוסט אמיטר – מתודה tost פונקציית ההפעלה.

אנו חייבים את שניהם על מנת שתופיע ההודעה.

הבחירה היכן למקם את ההגדרות נמצאת בידנו.

ניתן ליצור בתוך תיקיית services, קובץ service להודעות ולהגדרות ולהשאיר את הקריאה ל-tost והקונטיינר נקיים.

אנו נוסיף את הקומפוננטה ב- app.tsx מעל כל הקומפוננטות ובכך נוכל להשתמש בה בשאר הקומפוננטות.

ה-tost מקבל string עם תוכן ההודעה ואובייקט עם הגדרות (אופציות).

ניצור פונקציה שתפעיל את tost.

נפעיל את הפונקציה בתוך ה- onSubmit

נקרא לקומפוננטה tost ב-app.tsx ובכך כל הקומפוננטות יקבלו גישה אליו. (מחוץ לרואטרים).

קובץ ההגדרות service

```
import { Flip, toast } from "react-toastify";

export function successMessage(massage: string) {
  toast.success(massage, {
    position: "top-left",
    autoClose: 2000,
    theme: "colored",
    transition: Flip,
  });
}

export function errorMessage(massage: string) {
  toast.error(massage, {
    position: "top-left",
    autoClose: 5000,
    theme: "dark",
    transition: Flip,
  });
}
```

בקובץ app.tsx נייבא את הקומפננטה ונמקם אותה בתוך ה-return ומחוץ לראוטרים:

</ ToastContainer>

```
import { ToastContainer } from "react-toastify";
```

בקומפונטה שבה נרצה להציג את ההודעות נייבא את המתודות שיצרנו:

```
import { errorMessage, successMessage } from
"../services/feedbackService";
```

ונפעיל את ההודעות successMessage, errorMessage במיקומים הרלוונטים לדוגמא:

```
<i
  className="fa-solid fa-user-xmark text-danger pointer"
  onClick={() => {
    if (window.confirm("Are you sure?")) {
      deleteCustomer(customer.id as string)
        .then(() => {
          successMessage(`${customer.email} Deleted successfuly`);
          setCustomersChanged(!customersChanged);
        })
        .catch((err) => {
          errorMessage("Error, user not deleted. Please try again...");
          console.log(err);
        });
    }
```

```
  }}></i>;
```

דוגמא נוספת - הפעלת הפונקציה ב-onSubmit של formik:

```
onSubmit: (values) => {
      addCustomer(values as Customer);
      successMessage("Customer was added successfuly");
      navigate("/");
```

useRef

מלשון reference (הפניה)

משתנה אמצע שיבצע עדכון בדפדפן ולא יבצע רנדור מחדש של הקומפוננטה.

רנדר כפי שכבר הבנו זהו עיבוד מחדש של הקוד.

משתנים	עדכון ב-DOM של הדפדפן	רינדור
רגיל	X	X
useState	V	V
useRef	V	X

כפי שאנחנו רואים בטבלה:

משתנה רגיל - אינו מעדכן את ה-DOM בדפדפן ואינו מבצע רנדור מחדש של הקומפוננטה.

useState - מעדכן את ה-DOM של הדפדפן ומבצע רנדור מחדש של הקומפונטטה.

useRef – מעדכן את ה-DOM של הדפדפן ואינו מבצע רינדור מחדש של הקומפוננטה.

ניתן להשתמש ב-useRef לעדכון איזשהו רפרנס.

כאשר נרצה לגשת לערך נצטרך לגשת לאובייקט נקודה current.

בואו נראה דוגמא להצגה של מספר השינויים באינפוט מסוים בתוך תגית פיסקה:

```
 import {
  FunctionComponent,
  useContext,
  useEffect,
  useRef,
  useState,
} from "react";
import { ThemeContext, ThemeContextType } from
"../context/ThemeContext";

interface HomeProps {}

const Home: FunctionComponent<HomeProps> = () => {
  const themeContext = useContext(ThemeContext) as ThemeContextType;
```

```tsx
  const { theme } = themeContext;

  let [name, setName] = useState<string>("");
  let counter = useRef<number>(1);
  console.log(counter);

  useEffect(() => {
    counter.current = counter.current + 1;
  });

  return (
    <div className={`theme-${theme}`}>
      <p className={`text-warning t-${theme}`}>Theme: {theme}</p>
      <p>App renderd {counter.current} times</p>
      <p>{name}</p>
      <input
        type="text"
        onChange={(e) => {
          setName(e.target.value);
        }}
      />
    </div>
  );
};

export default Home;
```

כאן יצרנו למעשה משתנה קאונטר מסוג useRef שהערך ההתחלתי שלו 1.

הגדרנו פונקציה useEffect שתפעל בכל רינדור.

הגדרנו גם name מסוג useState.

בשדה input יש לנו פונקציה onChange שמפעילה את setName עם הערך של השדה.

למעשה כל שינוי בשדה אינפוט מפעיל את הפונקציה setName שמפעילה את useEffect שמגדילה את הקאונטר שלנו ב-1.

הקאונטר אינו מביא לידי רינדור מחדש של הקומפוננטה אלא רק מעדכן בדפדפן את הערך שלו.

את הערך העדכני של הקאונטר הצגנו בתגית פיסקה בדפדפן.

useRef הוא אחד מה-Hooks (ווית) השימושיים ביותר ב-React. הוא מספק דרך "לשמור" ערך שאינו גורם לעדכון מחדש של הקומפוננטה כאשר הוא משתנה. במילים אחרות, זהו כלי שמאפשר לנו לעבוד עם ערכים שאינם חלק מה-state או מה-props של הקומפוננטה.

למה משתמשים ב-useRef?

ישנן מספר סיבות עיקריות לשימוש ב-useRef:

- גישה לאלמנטים ב-DOM: אחת השימושים הנפוצים ביותר של useRef הוא גישה ישירה לאלמנטים ב-DOM. לדוגמה, אם נרצה למקד את הסמן (focus) על שדה טקסט מסוים לאחר שהקומפוננטה עלתה, נוכל להשתמש ב-useRef כדי לקבל הפניה לאלמנט ה-input ולבצע עליו את הפעולה focus.
- שמירת ערכים משתנים: useRef יכול לשמש גם לשמירת ערכים שמשתנים במהלך חיי הקומפוננטה, אך אינם מחייבים עדכון מחדש שלה. לדוגמה, נוכל להשתמש ב-useRef כדי לספור כמה פעמים הקומפוננטה רנדרה, מבלי לגרום לעדכון חוזר בכל פעם שהערך מתעדכן.
- יצירת הפניות (Refs) לאלמנטים: useRef מאפשר לנו ליצור הפניות (Refs) לאלמנטים ב-DOM, אותן נוכל להעביר הלאה לקומפוננטות אחרות. זה שימושי כאשר אנו רוצים לאפשר לקומפוננטה חיצונית לשלוט על אלמנטים בתוך קומפוננטה אחרת.

כיצד להשתמש ב-useRef?

כדי להשתמש ב-useRef, עלינו לייבא אותו מהספרייה 'react' ולקרוא לו בתוך הקומפוננטה שלנו. הפונקציה useRef מקבלת כפרמטר את הערך ההתחלתי של ה-ref ומחזירה אובייקט עם מאפיין בשם current, שמכיל את הערך הנוכחי של ה-ref.

```
import React, { useRef, useEffect } from "react";

function MyComponent() {
  const inputRef = useRef(null);

  useEffect(() => {
    // לאחר שהקומפוננטה עלתה, נפנה את הסמן לשדה הטקסט
    inputRef.current.focus();
  }, []);

  return (
    <div>
      <input type="text" ref={inputRef} />
    </div>
  );
}
```

בדוגמה זו, יצרנו ref בשם inputRef ונתנו לו ערך התחלתי null. לאחר מכן, השתמשנו ב-useEffect כדי לגשת לאלמנט ה-input באמצעות inputRef.current ולמקד עליו את הסמן.

חשוב לזכור:

שינוי הערך של current לא גורם לעדכון מחדש של הקומפוננטה.

אין לגשת לערך של current במהלך הרינדור, אלא רק לאחר מכן (לדוגמה, בתוך useEffect או event handler).

useContext

מאפשר לנו להעביר מידע בין קומפוננטות.

אנחנו מגדירים לו גבולות גזרה אילו קומפוננטות יוכלו לגשת למידע, באמצעות המילה - provider.

לדוגמא מקובץ app.tsx כאשר הקומפוננטה שלנו ThemeContext עוברת לשאר הקומפוננטות שנמצאות בתוך ה-provider שלנו עם תגית פותחת וסוגרת:

```
return (
  <>
    <Transition />
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      <ColorThemeToggle />
      <Home />
      <ColorChange />
      <ColorChangeSep />
    </ThemeContext.Provider>
    <Reducer />
  </>
);
```

בואו נראה את קובץ ההגדרות ThemeContext.ts וכיצד הגדרנו אותו:

```
import { createContext } from "react";

export type Theme = "light" | "dark";

export interface ThemeContextType {
  theme: Theme;
  toggleTheme: () => void;
}

export const ThemeContext = createContext<ThemeContextType |
undefined>(
  undefined
);
```

createContext – יוצרת קונטקסט חדש ומאחסנת מידע במשתנה.

בקובץ app.tsx יצרנו את ההגדרות הבאות מעל ה-return:

```
const [theme, setTheme] = useState<Theme>("light");
const toggleTheme = () => setTheme(theme === "light" ? "dark" :
"light");
```

useContext – שימוש במידע קיים.

אנחנו מקבלים בחזרה אובייקט המכיל את המידע + פונקציית ההפעלה.

בקומפוננטה אחרת שלנו למשל colorChange הגדרנו את הדבר הבא:

```
const themeContext = useContext(ThemeContext) as ThemeContextType;
const { theme } = themeContext;
```

*יש לשים לב למשתנה עם אות ראשית t קטנה שמפעיל את useContext שמקבל כערך את הThemContext מקובץ ההגדרות.

ניתן להשתמש במספר קונטקסטים בפרויקט אחד לכן לכל קונטקסט נבחר שם בעל משמעות כדי שנוכל לגשת אליו בעזרת useContext.

useContext מחזיר אובייקט עם המידע שהוא מכיל.

על מנת להשתמש ב-useContext נצטרך:

1. ליצור את הקונטקסט עצמו createContext
2. נייבא את המידע שהוא מכיל ע"י useContext
3. בקובץ ה-app.tsx נשים את הקומפונטה שמשתמש ב-useContext בתוך גבולות הגזרה של ה-provider.

כפתור לשינוי מצב תצוגה בהיר/כהה

שימוש במידע המועבר מהקומפוננטה הראשית app לקומפוננטות אחרות באפליקציה.

- ניצור תיקיה חדשה context
- ניצור קובץ הגדרות חדש themeContext.ts
- בתוכו ניצור קונטקסט createContext

```
import { createContext } from "react";

export type Theme = "light" | "dark";

export interface ThemeContextType {
  theme: Theme;
  toggleTheme: () => void;
}

export const ThemeContext = createContext<ThemeContextType |
undefined>(
  undefined
);
```

- בחלק הלוגי של קובץ app.tsx נגדיר state שיש לו ערך דיפולטיבי light
- ניצור פונקציה שתפקידה לשנות את המצב מבהיר לכהה והפוך

```
function App() {
  const [theme, setTheme] = useState<Theme>("light");
```

```
  const toggleTheme = () => setTheme(theme === "light" ? "dark" :
"light");
```

- ניצור קומפוננטה חדשה colorThemeToggle שתציג לנו את כפתור מצב התצוגה
- נגדיר באינפוט של הכפתור onChange שבכל שינוי יפעיל את הפונקציה שלנו לשינוי בין מצבי התצוגה שהגדרנו.

```
import { FunctionComponent, useContext } from "react";
import { ThemeContext, ThemeContextType } from
"../context/ThemeContext";
interface ColorThemeToggleProps {}

const ColorThemeToggle: FunctionComponent<ColorThemeToggleProps> = ()
=> {
  const themeContext = useContext(ThemeContext) as ThemeContextType;
  const { theme, toggleTheme } = themeContext;
  return (
    <>
      <div className="form-check form-switch">
        <input
          className="form-check-input"
          type="checkbox"
          role="switch"
          id="flexSwitchCheckDefault"
          onChange={toggleTheme}
        />
        <label className="form-check-label"
htmlFor="flexSwitchCheckDefault">
          Dark mode
        </label>
      </div>
    </>
  );
};

export default ColorThemeToggle;
```

- נבצע שינויים בקומפוננטת home שימוש בקונטקסט useContext כדי לייבא את המידע:

```
const Home: FunctionComponent<HomeProps> = () => {
  const themeContext = useContext(ThemeContext) as ThemeContextType;
  const { theme } = themeContext;
```

- ל-div הראשי נגדיר קלאס עם המפתח theme שהתקבל:

כעת יהיו לנו 2 מצבים לקלאסים אפשריים:

theme-light

theme-dark

```
return (
  <div className={`theme-${theme}`}>
    <p className={`text-warning t-${theme}`}>Theme: {theme}</p>
    <p>App renderd {counter.current} times</p>
    <p>{name}</p>
    <input
      type="text"
      onChange={(e) => {
        setName(e.target.value);
      }}
    />
  </div>
);
```

- נגדיר את גבולות הגיזרה שלנו לאילו קומפוננטות יועבר המידע באמצעות provider, וניתן להם ערך value עם המידע השמור באובייקט:

```
<ThemeContext.Provider value={{ theme, toggleTheme }}>
  <ColorThemeToggle />
  <Home />
  <ColorChange />
  <ColorChangeSep />
</ThemeContext.Provider>;
```

- ניתן להגדיר גם קלאס מיוחד לעיצוב פיסקה, כך יהיו לנו 2 מצבים לקלאסים אפשריים:

t-light

t-dark

```
<p className={`text-warning t-${theme}`}>Theme: {theme}</p>;
```

כעת נוכל להגדיר את העיצוב ב-CSS.

useReducer

האב הקדמון של useState.

useState יודעת לקבל גם פונקציה.

ה-reducer הוא הוק שבאמצעותו אנחנו יכולים לנהל state בצורה מורכבת יותר.

פונקצית reducer מקבלת פעולה מעבדת אותה ומחזירה state חדש.

useReducer הוא Hook עוצמתי ב-React המאפשר ניהול מורכב של state בתוך קומפוננטה. הוא שימושי במיוחד כאשר יש לוגיקה מורכבת לעדכון הstate או כאשר הstate מורכב ממספר חלקים.

הבנת useReducer

<u>useReducer מקבל שני ארגומנטים:</u>

reducer: פונקציה שמקבלת את הstate הנוכחי ואובייקט action, ומחזירה state חדש.

initialState: הערך ההתחלתי של הstate.

<u>useReducer מחזיר שני ערכים:</u>

state: הערך הנוכחי של הstate.

dispatch: פונקציה שמשמשת לשליחת action לreducer.

ה-reducer הוא פונקציה שמקבלת שני ארגומנטים:

state: הstate הנוכחי.

action: אובייקט שמכיל מידע על הפעולה שצריך לבצע על הstate.

ה-**reducer** צריך להחזיר state חדש בהתאם ל-action.

ה-**action** הוא אובייקט שמכיל מידע על הפעולה שצריך לבצע על הstate. בדרך כלל הוא מכיל לפחות שני מאפיינים:

type: מחרוזת שמזהה את סוג הפעולה.

payload (אופציונלי): מידע נוסף שנדרש לביצוע הפעולה.

```
import React, { useReducer } from "react";

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      return state;
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      Count: {state.count}
```

```
      <button onClick={() => dispatch({ type: "increment"
})}>Increment</button>
      <button onClick={() => dispatch({ type: "decrement"
})}>Decrement</button>
    </div>
  );
}
```

בדוגמה זו, ה-reducer מקבל את ה-state הנוכחי ואת ה-action. ה-action מכיל את סוג הפעולה ('increment' או 'decrement'). ה-reducer מחזיר state חדש בהתאם לסוג הפעולה.

בואו נפרק את סדר הפעולות ונראה אם ההבנה שלך נכונה:

onClick: כאשר כפתור נלחץ, הפונקציה onClick נקראת.

dispatch: הפונקציה onClick קוראת לפונקציה dispatch עם אובייקט action כארגומנט. אובייקט ה-action מכיל מידע על הפעולה שצריך לבצע, כמו { type: 'increment' }.

reducer: הפונקציה dispatch מעבירה את ה-action ל-reducer. ה-reducer הוא פונקציה שמקבלת שני ארגומנטים: ה-state הנוכחי ואת ה-action. ה-reducer מחליט כיצד לעדכן את ה-state בהתבסס על ה-action ומחזיר state חדש.

עדכון ה-React: state משתמש ב-state החדש שחזר מה-reducer כדי לעדכן את הקומפוננטה.

רינדור מחדש: הקומפוננטה מתרנדרת מחדש עם ה-state החדש, ושינויים מוצגים למשתמש.

useReducer "מחזיק" את ה-state. הוא גם מספק את הפונקציה dispatch, שמאפשרת לך לשלוח actions ל-reducer.

ה-reducer מקבל את ה-state הנוכחי ואת ה-action ומחזיר state חדש.

React משתמש ב-state החדש כדי לעדכן את הקומפוננטה.

מתי להשתמש ב-useReducer?

- כאשר יש לוגיקה מורכבת לעדכון הstate.
- כאשר ה-state מורכב ממספר חלקים.
- כאשר יש צורך לשמור על היסטוריה של ה-state.

טיפים

- הקפידו על כתיבת reducer פשוט וקל להבנה.
- השתמשו ב-action types קבועים כדי למנוע טעויות.
- חלקו את ה-reducer לפונקציות קטנות יותר במידת הצורך.

useTransition

מטרתו למנוע תקיעות בממשק צד הלקוח בזמן ביצוע פעולה כבדה.

באופן דיפולטיבי במידה ונפנה נניח לשרת לקבלת מידע כבד האפליקציה תיהיה בהמתנה לקבלת המידע ולא נוכל לבצע בה פעולות אחרות בזמן איסוף המידע, useTransition באה כדי למנוע מצבים כאלה ולאפשר לבצע פעולות ברקע מבלי לחסום את ממשק המשתמש.

המטרה – שיפור חווית המשתמש.

היא מונעת את הקפאת האפליקציה בכך שאנו תוחמים את העדכון הכבד וכביכול אומרים לו הפעולה/בקשה הזו היא בעדיפות נמוכה.

useTransition מחזיר:

- isPeding – משתנה בוליאני שמצביע על מצב הרנדור האם עדיין עובד ברקע כל עוד הוא עובד יחזיר true אחרת יחזיר false.
- startTransition – כל קוד שהוא עוטף יסומן כפעולה/בקשה בעדיפות נמוכה ואינו יחסום את ממשק המשתמש מביצוע פעולות אחרות במקביל.

אנחנו נשתמש בו עבור טעינת משאבים כבדים, על מנת לשפר חווית משתמש כי יכול להיות למשל שבזמן המידע הכבד נטען הגולש יתחרט וירצה לגשת למקום אחר באפליקציה והדבר יתאפשר לו במידה ועשינו שימוש ב-useTransition.

כל בקשה אחרת שתתקבל תיהיה חשובה יותר מהפעולה שעטופה ב-useTransition והאפליקציה תפעל כדי לבצע אותה. הבקשה בuseTransition היא בקשה משנית.

מתי להשתמש בהוק useTransition?

- פעולה שדורשת עדכון state עם מספר שינויים מאוד כבד.
- כשרוצים להפחית תקיעות ברקע בזמן ביצוע פעולות מאוד כבדות
- במצבים ספציפיים שגורמים לעומס על האפליקציה

תרגול עם json placeholder

נפתח קובץ חדש קומפוננטה transition.tsx

נגדיר useEffect שיפעיל fetch ל-API של ג'ייסון פלייסהולדר.

then. הראשון מקבל דטא

then. השני מבצע השמת של הדטא ל-state

catch. תופס את השגיאה וידפיס אותה בקונסול

פעולת ההשמה של הדטא היא כבדה לכן נשתמש ב-useTransition

מלבד isPeding ניתן להוסיף state נוסף למשל isLoading שייבצע של הערך false ל-state לאחר ביצוע השמה של הדטא.

לשם ההמחשה ניקח ספינר מספריית bootstrap על מנת לראות מצב בו המידע נטען.

נבצע מניפולציה למידע שהתקבל כדי להציג אותו בדפדפן באמצעות המתודה map.

נקרא לקומפוננטה transition בקומפוננטה הראשית app.tsx

isLoading – ירוץ מתחילת הקריאה ל-fetch

isPeding – ירוץ מהקריאה ל-startTransition