uploadInovice

🔹 חלק 1: ייבוא ספריות

```tsx
import { useState } from 'react';
import { fetchWithAuth } from '../utils/fetchWithAuth';
import '../styles/UploadInovice.css';
```

- `useState` – כדי לנהל את הקובץ, את מצב הטעינה, את תוצאת ה־OCR ואת הודעות השגיאה.
- `fetchWithAuth` – פונקציית עזר שכנראה מוסיפה את ה־JWT אוטומטית לקריאה (`Authorization` header).
- `UploadInovice.css` – העיצוב של הרכיב.

🔹 חלק 2: סטייטים (useState)

```tsx
const [file, setFile] = useState<File | null>(null);
const [ocrData, setOcrData] = useState<any>(null);
const [status, setStatus] = useState<'idle' | 'loading' | 'done' | 'error'>('idle');
const [error, setError] = useState('');
```

- `file` – הקובץ שהמשתמש מעלה.
- `ocrData` – הנתונים שחזרו מהשרת לאחר ניתוח ה־OCR.
- `status` – סטטוס כללי של הפעולה (רגיל, בטעינה, הצלחה, או שגיאה).
- `error` – טקסט של הודעת שגיאה אם משהו נכשל.

📌 התחביר הכללי של useState עם TypeScript:

```ts
const [value, setValue] = useState<type>(initialValue);
```

- `value` – המשתנה (מה שנשמר בסטייט).
- `setValue` – הפונקציה שמשנה את הערך.
- `<type>` – **הטיפוס** של המשתנה (TypeScript).
- `initialValue` – הערך ההתחלתי של הסטייט.

1. `const [ocrData, setOcrData] = useState<any>(null);`

- את יוצרת משתנה בשם `ocrData` שיכול להכיל **כל מבנה של נתונים שמתקבל מהשרת**.
- `any` = כל דבר – אין הגבלה על הטיפוס.
- את מתחילה עם `null`, כלומר: **עוד לא התקבלו נתונים**.

💡 אם היית יודעת מה מבנה הנתונים, עדיף להשתמש בטיפוס מותאם (ולא `any`).
לדוגמה:

Edit Copy ts

```ts
type OcrData = {
  invoiceNumber: string;
  totalWithVat: number;
  invoiceDate: string;
  // וכו'
};
const [ocrData, setOcrData] = useState<OcrData | null>(null);
```

2. `const [file, setFile] = useState<File | null>(null);`

- את יוצרת משתנה בשם `file` שמיועד להכיל **קובץ**.
- `File` זה טיפוס מובנה בדפדפן של אובייקט קובץ (מ־`<input type="file">`)
- את מתחילה עם `null`, כלומר – **עוד לא נבחר קובץ**.

> כשמישהו בוחר קובץ בטופס, את קוראת:

Edit Copy tsx

```tsx
onChange={(e) => setFile(e.target.files?.[0] || null)}
```

וזה מכניס את הקובץ הראשון שהועלה לתוך `file`.

🔹 חלק 3: הפונקציה `handleUpload`

```tsx
const handleUpload = async () => {
```

זוהי הלוגיקה שמופעלת כאשר המשתמש לוחץ על כפתור "**העלה וסרוק**".

פירוט כל השלבים:

🔸 1. בדיקה אם נבחר קובץ

```tsx
if (!file) {
  setError('נא לבחור קובץ');
  return;
}
```

🔸 2. איפוס נתונים והכנה להעלאה

```tsx
setStatus('loading');
setError('');
setOcrData(null);
```

🔸 3. יצירת FormData עם הקובץ

```tsx
const formData = new FormData();
formData.append('file', file);
```

> 📦 FormData מאפשר לשלוח קבצים ב־POST בצורה נוחה.

💡 מה זה `FormData`?

`FormData` היא מחלקה (class) ב־JavaScript/TypeScript שנועדה ליצירת **אובייקטים שמדמים טופס HTML**, כדי שאפשר יהיה לשלוח אותם בקלות לשרת – במיוחד כשיש בהם **קבצים**.

> במקום לשלוח JSON רגיל – משתמשים ב־`FormData` כדי **לשלוח קובץ אמיתי** (כמו `File` מתוך `input>` `type="file"<`).

🔍 למה לא שולחים קובץ עם JSON רגיל?

כי JSON הוא רק טקסט 📄

הוא **לא יכול להכיל קבצים** או מידע בינארי (כמו PDF, JPG וכו').

לעומת זאת – `FormData` **שולח את הקובץ כמו שהדפדפן שולח טופס רגיל**, עם `Content-Type: multipart/form-data`.

🔸 4. שליחת הבקשה לשרת

```tsx
const res = await fetchWithAuth('https://localhost:7129/api/Upload/upload', {
  method: 'POST',
  body: formData,
});
```

`fetchWithAuth` מוסיף אוטומטית טוקן JWT ל־headers (בנית: `Authorization: Bearer ...`)

🔸 5. טיפול בשגיאה מהשרת

```tsx
if (!res.ok) {
  const msg = await res.text();
  throw new Error(`שגיאה בהעלאת קובץ ל-OCR: ${msg}`);
}
```

🔸 6. קבלת הנתונים מהשרת

```tsx
const data = await res.json();
setOcrData(data);
setStatus('done');
```

🔸 7. טיפול בשגיאה כללית (תקלה ברשת/שרת)

```tsx
catch (err: any) {
  console.error(err);
  setError(err.message || 'שגיאה כללית');
  setStatus('error');
}
```

📂 בחירת קובץ:

```tsx
<label className="custom-file-upload">
  בחרי קובץ
  <input
    type="file"
    accept="image/*,application/pdf"
    onChange={(e) => setFile(e.target.files?.[0] || null)}
  />
</label>
```

- `accept` מגביל לקבצי תמונה או PDF
- הקובץ נכנס לסטייט `file`

⬆️ כפתור העלאה:

```tsx
<button
  onClick={handleUpload}
  disabled={status === 'loading'}
  className="upload-button"
>
  {status === 'loading' ? 'העלה וסרוק' : 'מעלה וסורק...'}
</button>
```

- אם `status === 'loading'`, הכפתור מוצג במצב טעינה.
- אחרת – הכפתור זמין ללחיצה.

🔍 תצוגה של תוצאת ה־OCR:

```tsx
{ocrData && (
  <div className="upload-preview">
    <h3 className="preview-title">חשבונית שנשמרה:</h3>
    <pre>{JSON.stringify(ocrData, null, 2)}</pre>
  </div>
)}
```

- מציג את כל המידע שחזר מהשרת (בד"כ כולל: תאריך, מספר חשבונית, שם עסק, סכום וכו').
- משתמש ב־`<pre>` כדי לשמור על עיצוב קריא.

🧠 איך תסבירי את זה בראיון?

> "הרכיב `UploadInvoice` מאפשר למשתמש להעלות קובץ תמונה או PDF של חשבונית.
> ברגע שהמשתמש בוחר קובץ ולוחץ על כפתור, אני שולחת את הקובץ לשרת עם FormData ו־JWT.
> השרת מבצע OCR ומחזיר נתונים – אני שומרת אותם בסטייט ומציגה אותם למשתמש.
> טיפלתי גם בשגיאות מהשרת או מהרשת, ודאגתי לחוויית משתמש טובה כולל סטטוס טעינה והודעות שגיאה."

ExpensesTable

🔹 חלק 1: טיפוס `Expense`

Edit Copy tsx

```tsx
export type Expense = {
  id: number;
  businessName: string;
  invoiceNumber: string;
  businessId: string;
  documentType: string;
  invoiceDate: string;
  totalBeforeVat: number;
  totalWithVat: number;
  category: string;
};
```

- מגדיר את **המבנה של כל הוצאה** (Expense).
- זה מאפשר לנו להשתמש ב־TypeScript כדי לדעת מה יש בכל שדה ולמנוע טעויות.

🔹 חלק 2: useState – ניהול סטייטים

```tsx
const [expenses, setExpenses] = useState<Expense[]>([]);
```

- מערך של כל ההוצאות שנטענו מהשרת.

```tsx
const [searchTerm, setSearchTerm] = useState('');
const [selectedCategory, setSelectedCategory] = useState('');
const [minTotal, maxTotal, dateFrom, dateTo]...
```

- כל השדות כאן משמשים ל**סינון וחיפוש**.

✅ קוד עם פירוש שורה-שורה:

```tsx
const fetchExpenses = useCallback(() => {
```

🔹 אנחנו עוטפים את הפונקציה ב־`useCallback` כדי:

- לוודא שהיא **לא נבנית מחדש סתם** בכל רינדור (Performance optimization).
- זה חשוב כי אנחנו משתמשים בה בתוך `useEffect` – ושם רוצים למנוע לולאות אין־סופיות.

🔹 שלב 1: יצירת אובייקט פרמטרים (סינון)

Edit Copy tsx

```tsx
const params: any = {};
if (searchTerm) params.name = searchTerm;
if (selectedCategory) params.category = selectedCategory;
if (minTotal !== undefined && !isNaN(minTotal)) params.min = minTotal;
if (maxTotal !== undefined && !isNaN(maxTotal)) params.max = maxTotal;
if (dateFrom) params.from = dateFrom;
if (dateTo) params.to = dateTo;
```

💡 מה קורה פה?

- נבנה אובייקט בשם `params` (כלומר: `{}`).
- כל תנאי מוסיף לתוכו **פרמטר רק אם המשתמש מילא אותו**.
- לדוגמה, אם המשתמש כתב "שופרסל" – נשלח `params.name = "שופרסל"`.

> כך נוצר קישור כזה לשרת (לדוגמה):

Edit Copy arduino

```arduino
/api/expenses?name=שופרסל&category=מזון&min=100&max=500&from=2023-01-01
```

🔹 שלב 2: הפעלת מצב טעינה

```tsx
setLoading(true);
```

- זה משנה את `loading` ל־`true` כדי להציג הודעת "טוען נתונים..." במסך.

🔹 שלב 3: שליחת הבקשה לשרת עם axios

```tsx
axios.get('https://localhost:7129/api/expenses', {
  params,
  headers: {
    Authorization: `Bearer ${localStorage.getItem('token')}`
  }
})
```

- `axios.get` שולח בקשה לכתובת שצוינה.
- `params` = הפרמטרים של הסינון.
- `headers` = כולל את הטוקן של המשתמש כדי שהשרת יזהה אותו (`JWT` מ־localStorage).

🔹 שלב 4: טיפול בתשובה

```tsx
.then(res => {
  setExpenses(res.data);
  setLoading(false);
})
```

- אם הבקשה הצליחה:
- הנתונים שהתקבלו (`res.data`) מוכנסים למשתנה `expenses`.
- מצב הטעינה מתבטל.

🔹 שלב 5: טיפול בשגיאה

```tsx
.catch(err => {
  console.error("❌ שגיאה:", err);
  setLoading(false);
});
```

- אם הייתה שגיאה בבקשה (שרת לא עונה, טוקן לא תקף וכו'):
- מדפיסים שגיאה לקונסול.
- מסיימים את מצב הטעינה.

🔄 הערך של `[... ,searchTerm, selectedCategory]`

זוהי רשימת התלויות של `useCallback`.

כלומר: **אם אחת מהן משתנה** – `fetchExpenses` תיבנה מחדש עם הערכים החדשים.

🧠 סיכום פשוט:

הפונקציה `fetchExpenses` בונה את כל הפרמטרים לסינון על בסיס הקלט של המשתמש,
שולחת בקשת GET לשרת עם טוקן אימות,
ואם הצליח – היא שומרת את התוצאות במשתנה `expenses` ומפסיקה את מצב הטעינה.

הסינון מתבצע בצד השרת –והקליינט (הפרונט) רק שולח את **פרמטרי הסינון** בתוך קריאת ה־GET.

✅ חלק 1: מחיקת הוצאה

```tsx
const deleteExpense = (id: number) => {
  if (!window.confirm('האם למחוק את ההוצאה?')) return;
  axios.delete(`https://localhost:7129/api/expenses/${id}`)
    .then(() => fetchExpenses())
    .catch(err => console.error('❌ שגיאת מחיקה:', err));
};
```

מה קורה פה?

1. כשמשתמש לוחץ על 🗑 – נשאלת שאלה: "האם למחוק?"
2. אם כן – נשלחת קריאת `DELETE` לשרת עם מזהה ההוצאה.
3. אחרי הצלחה → קוראים ל־`fetchExpenses()` כדי לרענן את הטבלה.
4. אם יש שגיאה – מדפיסים אותה לקונסול.

מה קורה כשמישהו לוחץ על כפתור "ערוך"?

הפונקציה הזו מופעלת:

```tsx
const startEdit = (exp: Expense) => {
  setEditingId(exp.id);
  setEditedExpense({ ...exp });
};
```

שורה שורה:

- `setEditingId(exp.id)`
  ⏩ אנחנו שומרים את ה־ID של ההוצאה הזו כדי לדעת שהיא **השורה שנמצאת בעריכה**.
- `setEditedExpense({ ...exp })`
  ⏩ אנחנו מעתיקים את כל הנתונים של ההוצאה אל תוך משתנה חדש `editedExpense`, שבו נשמור **שינויים** כשהמשתמש מתחיל להקליד.

> כך React יודע: השורה עם `id === editingId` → תציג `<input>` במקום טקסט רגיל.

ואז בקוד התצוגה (`render`), קורה הקסם הזה:

```tsx
{editingId === e.id ? (
  <input value={editedExpense.businessName} onChange={...} />
) : (
  e.businessName
)}
```

- אם השורה נמצאת בעריכה – תוצג שדה טקסט.
- אם לא – יוצג הטקסט הרגיל.

💾 חלק 3 – שמירת עריכה

נגיד שהמשתמש משנה את "שופרסל" ל-"AM:PM", ואז לוחץ על כפתור 💾

זה מפעיל את הפונקציה:

Edit Copy tsx

```tsx
const saveEdit = () => {
  if (!editingId) return;

  axios.put(`https://localhost:7129/api/expenses/${editingId}`, editedExpense)
    .then(() => {
      fetchExpenses();
      cancelEdit();
    })
    .catch(err => console.error('❌ שגיאת עדכון:', err));
};
```

שורה שורה:

1. `if (!editingId) return;`
   ⏩ אם לא בעריכה – פשוט יוצאים.
2. `axios.put(...)`
   ⏩ שולחים בקשת PUT לשרת עם:
   - ה־ID של ההוצאה (`editingId`)
   - כל הנתונים שהמשתמש ערך (`editedExpense`)

   📩 לדוגמה, אם המשתמש שינה את שם העסק והסכום – זה מה שישלח לשרת.
3. `then(() => { fetchExpenses(); cancelEdit(); }).`
   ⏩ אם השמירה הצליחה:
   - מרעננים את הטבלה (`fetchExpenses`)
   - יוצאים ממצב עריכה (`cancelEdit()`)

🚫 מה עושה `cancelEdit`?

```tsx
const cancelEdit = () => {
  setEditingId(null); // מפסיק את מצב העריכה
  setEditedExpense({}); // מנקה את השינויים
};
```

דוגמה מלאה מהחיים:

לפני לחיצה על עריכה:

- מופיע טקסט: "שופרסל", 500₪

אחרי לחיצה על ✏️:

- מופיע input עם ערך "שופרסל"
- המשתמש משנה ל־"AM:PM"
- לוחץ על 💾

התוצאה:

- נשלחת בקשת PUT לשרת עם הנתונים החדשים
- השרת שומר במסד הנתונים
- הטבלה מתעדכנת עם הנתון החדש

💬 הסבר בראיון:

> "ברגע שהמשתמש לוחץ על כפתור עריכה, אני שומרת את מזהה ההוצאה בעריכה ואת הנתונים שלה למשתנה `editedExpense`.
> בטבלה, אם שורה נמצאת בעריכה – מציגים לה `input`.
> המשתמש משנה את הערכים, ולחיצה על שמירה שולחת קריאת PUT לשרת עם הנתונים החדשים.
> לאחר הצלחה, אני טוענת מחדש את ההוצאות ויוצאת ממצב עריכה."

```tsx
useEffect(() => {
  fetchExpenses();
}, [fetchExpenses]);
```

🎯 מה זה עושה?

- מריץ את `fetchExpenses()` **ברגע שהקומפוננטה נטענת** בפעם הראשונה.
- זו **הטעינה הראשונית** של כל ההוצאות מהשרת.

```tsx
useEffect(() => {
  const debounce = setTimeout(() => {
    fetchExpenses();
  }, 500);
  return () => clearTimeout(debounce);
}, [fetchExpenses]);
```

🎯 מה זה עושה?

- עוקב אחרי כל שינוי בתנאי הסינון:
- חיפוש לפי שם (`searchTerm`)
- קטגוריה
- סכום מינימום/מקסימום
- תאריך מ-ועד
- ואז מחכה **500ms (חצי שנייה)** לפני שמבצע קריאת `fetchExpenses()` .

אם בינתיים המשתמש ממשיך להקליד או לשנות משהו → מבטלים את הקריאה הקודמת (clearTimeout).

זה מונע מצב שבו **כל תו שהמשתמש מקליד שולח בקשת API לשרת**.

לדוגמה:

- אם המשתמש מקליד "שופרסל", בלי debounce זה היה שולח 6 קריאות:
- `ש` , `שו` , `שופ` , `שופר` , `שופרס` , `שופרסל` ...

עם debounce של 500ms:

- הקריאה תישלח רק אחרי שהמשתמש סיים להקליד והפסיק לגעת לשנייה.

זה על הטבלת הוצאות על ה HTML

"בתוך רכיב טבלת ההוצאות יצרתי גם UI נוח לסינון חכם לפי קטגוריות, סכומים ותאריכים,
וגם ממשק עריכה ומחיקה של כל שורה בטבלה.
השתמשתי ב־state לניהול השורה שנמצאת בעריכה, וב־inputים שמופיעים דינאמית לפי ה־id.
כל שינוי נשלח לשרת עם PUT, וכל מחיקה עם DELETE.
אני משתמשת גם בחיווי טעינה ו־UX ברור כדי להבטיח שהמשתמש מבין בדיוק מה קורה."

💬 משפט מוכן לראיון:

> "בקומפוננטת `ExpensesTable`, בניתי ממשק מלא להצגת הוצאות עם סינון חכם לפי שם עסק, קטגוריה, סכומים ותאריכים.
> התממשקתי לשרת עם axios ו־JWT, תמכתי בעריכת שורות בטבלה בזמן אמת, כולל שמירה ושליחת PUT לשרת.
> השתמשתי ב־debounce כדי למנוע בקשות מיותרות בזמן סינון, ודאגתי לחוויית משתמש ברורה עם מצב טעינה, תצוגת שגיאות ופעולות עריכה ומחיקה."

DASHBOARD

2. יצירת מצב (state) לניהול הטאב הנבחר:

Edit Copy tsx

```tsx
const [tab, setTab] = useState<'upload' | 'expenses' | 'charts'>('upload');
```

- ברירת מחדל: הטאב הנבחר הוא "העלאה".
- ערכים אפשריים:
- `'upload'`
- `'expenses'`
- `'charts'`

4. טאבים (כפתורי ניווט):

```tsx
<div className="dashboard-tabs">
  <button onClick={() => setTab('upload')} ...> העלאת חשבוניות </button>
  <button onClick={() => setTab('expenses')} ...> ניהול הוצאות </button>
  <button onClick={() => setTab('charts')} ...> גרפים וסטטיסטיקות </button>
</div>
```

- כל כפתור משנה את הערך של `tab`.
- ה־`active` class מתווסף רק לכפתור הנבחר – לעיצוב (סביר להניח קו תחתון / צבע).

5. תוכן משתנה לפי הטאב:

```tsx
<div className="dashboard-content">
  {tab === 'upload' && <UploadInvoice />}
  {tab === 'expenses' && <ExpensesTable />}
  {tab === 'charts' && <DashboardCharts />}
</div>
```

- מציג רק קומפוננטה אחת מתוך השלוש בכל זמן נתון:
- 📄 UploadInvoice – העלאת קבצים
- 📋 ExpensesTable – טבלת הוצאות
- 📊 DashboardCharts – גרפים

> "בקומפוננטת Dashboard יצרתי ניווט פנימי בין שלושה טאבים עיקריים: העלאת חשבוניות, טבלת הוצאות וגרפים.
> השתמשתי ב־useState כדי לנהל את הטאב הנבחר, והצגתי את הקומפוננטה המתאימה בהתאם.
> כל אחד מהטאבים משתמש בקומפוננטות אחרות שנכתבו בנפרד – מה ששומר על סדר ותחזוקה קלה.
> זה מאפשר חוויית משתמש נעימה, ללא טעינה מחודשת של הדף."

"במקום לעבור עמוד, החלטתי לממש שלושה טאבים פנימיים עם useState בלבד.
זה מאפשר חוויית משתמש חלקה ומהירה – בלי צורך בריענון הדף או ב־URL שונה.
זו גישה שמתאימה במיוחד למערכות ניהול, שבהן יש רכיב מרכזי שמכיל כמה אזורי תוכן נפרדים."

🔹 1. ייבוא ספריות:

```tsx
import { useEffect, useRef, useState } from 'react';
import {
  PieChart, Pie, Cell, Tooltip, LineChart, Line, XAxis, YAxis,
  CartesianGrid, Legend, ResponsiveContainer
} from 'recharts';
import { fetchWithAuth } from '../utils/fetchWithAuth';
import html2canvas from 'html2canvas';
import '../styles/DashboardCharts.css';
```

- ייבוא של React Hooks (`useState`, `useEffect`, `useRef`)
- `recharts` – ספרייה להצגת גרפים
- `html2canvas` – הופכת רכיב HTML לתמונה PNG
- `fetchWithAuth` – קריאה עם JWT

🔹 2. טיפוס המידע שמגיע מהשרת:

```ts
type SummaryResponse = {
  byCategory: { category: string; total: number }[];
  byMonth: { month: number; total: number }[];
};
```

- הנתונים הם אובייקט עם:
- `byCategory`: הוצאות לפי קטגוריה
- `byMonth`: הוצאות לפי חודש

🔹 3. useState ו־useRef

```tsx
const [summary, setSummary] = useState<SummaryResponse | null>(null);
const [loading, setLoading] = useState(true);
const pieRef = useRef<HTMLDivElement>(null);
const lineRef = useRef<HTMLDivElement>(null);
```

- `summary` – הנתונים שהגיעו מהשרת
- `loading` – האם הנתונים עוד נטענים
- `pieRef`, `lineRef` – כדי להוריד את הגרפים כתמונה (PNG)

🔹 4. טעינת הנתונים מהשרת (useEffect)

Edit Copy tsx

```tsx
useEffect(() => {
  fetchWithAuth('https://localhost:7129/api/Reports/summary')
    .then(res => res.json())
    .then(data => {
      setSummary(data);
      setLoading(false);
    })
    .catch(err => {
      console.error('שגיאה בטעינת הדוחות', err);
      setLoading(false);
    });
}, []);
```

- קורא לשרת את הסיכום (`api/Reports/summary/`)
- שומר את התוצאה ב־`summary`
- מדליק/מכבה את `loading`

🔹 5. פונקציה להורדת הגרף כ־PNG

Edit Copy tsx

```tsx
const downloadChart = async (ref, name) => {
  if (!ref.current) return;
  const canvas = await html2canvas(ref.current);
  const link = document.createElement('a');
  link.download = `${name}.png`;
  link.href = canvas.toDataURL();
  link.click();
};
```

- מקבלת רפרנס לרכיב הגרפי (`pieRef`, `lineRef`)
- יוצרת קובץ תמונה מה־HTML
- שומרת למחשב בשם שהוגדר

🎨 תצוגה בפועל:

🔵 גרף עוגה – הוצאות לפי קטגוריה:

```tsx
<PieChart>
  <Pie data={summary.byCategory} ...>
    <Cell fill={colors[index % colors.length]} />
  </Pie>
  <Tooltip />
  <Legend />
</PieChart>
```

- `Pie` מציג עיגול שמחולק לקטגוריות
- `Cell` צובע כל פרוסה בצבע שונה
- `Tooltip` מציג נתונים בהצבעה
- `Legend` – מקרא

📍 עטוף ב־`ResponsiveContainer` – מותאם לרוחב המסך.

📈 גרף קו – הוצאות לפי חודש:

```tsx
<LineChart data={summary.byMonth}>
  <CartesianGrid />
  <XAxis dataKey="month" tickFormatter={...} />
  <YAxis />
  <Tooltip />
  <Legend />
  <Line dataKey="total" stroke="#00bfff" />
</LineChart>
```

- ציר X לפי מספר חודש → ממירים אותו לשם עברי (ינואר, פברואר וכו')
- `Line` – קו המחבר בין הסכומים בכל חודש
- `Tooltip` ו־`Legend` להשלמה גרפית

🖱️ כפתור הורדה לכל גרף:

```tsx
<button onClick={() => downloadChart(pieRef, 'expenses-by-category')}>
  הורד גרף כ־PNG
</button>
```

🧠 תשובה לראיון – ניסוח:

> "יצרתי קומפוננטת DashboardCharts שמציגה דו"ח ויזואלי של ההוצאות לפי קטגוריה וחודש.
> הנתונים נשלפים מהשרת, מעובדים ומוצגים באמצעות ספריית Recharts.
> בנוסף, אפשר להוריד כל גרף כקובץ PNG באמצעות html2canvas.
> הקפדתי על התאמה לרספונסיביות, טקסטים בעברית ותמיכה בחוויית משתמש נוחה וברורה."

📦 שם הקובץ: `fetchWithAuth.ts`

🎯 המטרה:

ליצור פונקציה כללית שמשמשת במקום `fetch`, ותמיד אוטומטית מוסיפה **Authorization: Bearer <token>** לכל בקשה – אם קיים טוקן.

🔍 ניתוח שורה שורה:

```ts
export async function fetchWithAuth(
  url: string,
  options: RequestInit = {}
): Promise<Response> {
```

- פונקציה אסינכרונית
- מקבלת:
- `url` – הכתובת של ה־API
- `options` – האופציות של הבקשה (method, headers, body...)
- מחזירה `Promise<Response>` כמו `fetch` רגיל

```ts
const token = localStorage.getItem('token');
```

- שולפת את ה־JWT שנשמר בזיכרון המקומי לאחר ההתחברות.
- אם אין טוקן – פשוט לא מוסיפים Authorization

```ts
const headers = new Headers(options.headers || {});
```

- יוצרים אובייקט `Headers` חדש.
- אם ב־options כבר היו headers (כמו `Content-Type`), משאירים אותם.

```ts
if (token) {
  headers.set('Authorization', `Bearer ${token}`);
}
```

- מוסיפים את הכותר Authorization רק אם יש טוקן

```ts
return fetch(url, {
  ...options,
  headers,
});
```

- שולחים את הבקשה בעזרת `fetch`
- עם כל האפשרויות המקוריות + ה־headers המעודכנים

🧠 חוסך חזרתיות:

במקום לכתוב כל פעם:

```ts
fetch('/api/something', {
  headers: {
    'Authorization': 'Bearer ...'
  }
})
```

את פשוט עושה:

```ts
fetchWithAuth('/api/something')
```

🔐 שומר על אבטחה:

- אם שכחת להוסיף טוקן – זה כבר בפנים
- מרוכז במקום אחד → קל לשפר או לשנות בעתיד

"יצרתי פונקציית עזר `fetchWithAuth` שמוסיפה באופן אוטומטי את ה־JWT ל־headers בכל קריאה ל־API.
זה חוסך חזרתיות בקוד, מבטיח שכל הקריאות מאומתות, ומרוכז במקום אחד שניתן לשלוט בו."

APP.tsx

✨ מבנה הקובץ בקצרה:

```tsx
<AuthProvider>
  <BrowserRouter>
    <AppContent />
  </BrowserRouter>
</AuthProvider>
```

- `AuthProvider` – מנהל את המשתמש המחובר והטוקן.
- `BrowserRouter` – מאפשר ניווט בין עמודים.
- `AppContent` – מכיל את הלוגיקה של הניתוב + תצוגת Header.

🔐 רכיב `RequireAuth`

```tsx
function RequireAuth({ children }: { children: React.ReactNode }) {
  const token = localStorage.getItem('token');
  return token ? <>{children}</> : <Navigate to="/login" />;
}
```

> 🎯 רכיב שמוודא שיש טוקן (משתמש מחובר).
> אם כן – מציג את התוכן.
> אם לא – מעביר אוטומטית לעמוד `login/`.

🔒 זה חשוב כדי למנוע גישה לעמודים מוגנים מה-URL.

💡 `AppContent`

```tsx
function AppContent() {
  const location = useLocation();
  const { username, isLoading } = useAuth();

  const hideHeaderPaths = ['/login', '/register'];
  const showHeader = !isLoading && username && !hideHeaderPaths.includes(location.pathname);
```

- מקבלים את מיקום ה־URL הנוכחי.
- מוודאים שה־Header יוצג **רק אם המשתמש מחובר**, והעמוד **אינו** login/register.

> ⚠️ נותן לך שליטה חכמה בתצוגה – לפי המצב והעמוד.

🔽 שליטה על הצגת ה־Header:

```tsx
const hideHeaderPaths = ['/login', '/register'];
const showHeader = !isLoading && username && !hideHeaderPaths.includes(location.pathname);
```

> כאן מחליטים אם להציג את ה־Header או לא:

- לא מציגים אותו ב־ `/login` או `/register`
- כן מציגים אותו אם המשתמש מחובר

📌 כלומר: המשתמש רואה את ה־Header **רק כשהוא בדשבורד** ומחובר.

💡 הסבר פשוט:

> `App.tsx` מנהל את כל הניווט והגישה למערכת.
> הוא בודק אם המשתמש מחובר, מציג Header רק כשצריך, ומוודא שאין גישה לדשבורד בלי טוקן.
> הכול בנוי סביב קונטקסט של המשתמש ו־React Router.

✅ תיאור מרשים לראיון:

> "ב־ `App.tsx` עטפתי את האפליקציה עם `AuthProvider` שמנהל את מצב המשתמש, ו־ `BrowserRouter` לצורך ניווט.
> יצרתי רכיב `RequireAuth` שמגן על מסכים כמו `dashboard/` ודואג להחזיר ל־login אם המשתמש לא מחובר.
> בנוסף, שלטתי בתצוגה של ה־Header לפי מיקום בדף והאם המשתמש מחובר, כדי לייצר חוויית שימוש חכמה."

AuthContext.tsx A

1. טיפוס הקונטקסט:

```ts
type AuthContextType = {
  username: string | null;
  setUsername: (username: string | null) => void;
  logout: () => void;
  isLoading: boolean;
};
```

- את מגדירה מה כל קומפוננטה שתקבל את הקונטקסט תקבל בפועל:
- `username` – שם המשתמש הנוכחי
- `setUsername` – פונקציה לעדכון השם
- `logout` – פונקציה ל־logout
- `isLoading` – האם בטעינה

---

2. יצירת הקונטקסט:

```ts
const AuthContext = createContext<AuthContextType | undefined>(undefined);
```

> זהו ה־**context** עצמו – אפשר לקרוא לו "זיכרון גלובלי של התחברות".

---

3. `AuthProvider`

```ts
export const AuthProvider = ({ children }: { children: React.ReactNode }) => {
```

> זהו **רכיב עוטף** שנמצא ב־`App.tsx`, שמכניס את הקונטקסט לכל האפליקציה.

---

4. ניהול סטייט:

```ts
const [username, setUsername] = useState<string | null>(null);
const [isLoading, setIsLoading] = useState(true);
```

- `username` – ברירת מחדל: null (עוד לא התחברנו)
- `isLoading` – מתחיל ב־true, משתנה ל־false כשהבד[illegible] בוצעה

5. טעינה ראשונית עם useEffect:

Edit Copy ts

```ts
useEffect(() => {
  const saved = localStorage.getItem("username");
  if (saved) setUsername(saved);
  setIsLoading(false);
}, []);
```

> בודק אם יש שם משתמש ב־`localStorage` ומחזיר אותו לסטייט
> 📌 כך המשתמש נשאר מחובר גם אחרי רענון הדף.

---

6. פונקציית logout

Edit Copy ts

```ts
const logout = () => {
  localStorage.removeItem("token");
  localStorage.removeItem("username");
  setUsername(null);
  window.location.href = "/login";
};
```

> מוחקת את הטוקן ואת השם → מוחקת את ההתחברות → מחזירה לעמוד login
> 🔒 חשוב מאוד לשמירה על אבטחה.

7. ה־Provider עצמו:

```ts
<AuthContext.Provider value={{ username, setUsername, logout, isLoading }}>
  {children}
</AuthContext.Provider>
```

> מעביר לכל הילדים באפליקציה את הערכים והפונקציות שבקונטקסט
> כל קומפוננטה יכולה להשתמש בהם עם `useAuth()`!

---

8. ה־`useAuth` Hook

```ts
export const useAuth = () => {
  const context = useContext(AuthContext);
  if (!context) throw new Error("useAuth must be used within AuthProvider");
  return context;
};
```

> ייבוא נוח שאפשר להשתמש בו בכל מקום:

```tsx
const { username, logout, isLoading } = useAuth();
```

תשובה לראיון – בול בפוני:

> "יצרתי `AuthContext` שמנהל את כל המידע של המשתמש המחובר – כולל `username`, מצב טעינה ו־logout.
> עטפתי את כל האפליקציה ב־`AuthProvider` כדי שכל קומפוננטה תוכל לגשת ל־useAuth.
> ברגע שהמשתמש נכנס, אני שומרת את המידע ב־localStorage, וכך גם אחרי ריענון הדף הוא נשאר מחובר."