# **React Definitions**

**Library – ספריה**

קבוצה של פונקציות / מחלקות שניתן להשתמש בהן במערכת שאנו בונים. אנו קובעים מתי וכיצד להשתמש.

לדוגמה: jQuery.

**Framework**

מערכת סגורה שבתוכה אנו עובדים. היא לאחר מכן לוקחת את הקוד שלנו והופכת אותו למוצר המוגמר.

לדוגמה: Angular

**MPA – Multi Page Application**

זהו אתר המכיל מספר דפים פיזיים. כל דף בכתובת אחרת. ה-Frontend גולש לכל דף בנפרד הנמצא ב-Backend.

כך היו בנויים אתרים עד אזור 2010.

**SPA – Single Page Application**

זהו אתר המכיל רק דף פיזי אחד, לרוב בשם index.html. כל שאר ה"דפים" קיימים בתוך JavaScript ומוזרקים לאזור בתוך ה-index.html. זה אומר שהדפדפן גולש רק ל-index.html ולא לשאר הדפים. הדפדפן יכול לגלוש לשירות להחזרת מידע, אך הוא לא גולש להחזרת דף נוסף.

מ-2010 זו השיטה הפופולרית יותר לבניית אתרים.

**Component**

יחידת UI קטנה הניתנת להצבה על הדף בכל מקום וניתנת להצגה מספר פעמים.

זו הטכניקה הנכונה כיום לבניית אתרים. חלוקת ה-UI לחלקים קטנים והצבתם במקומות שונים על הדף.

קומפוננטה ב-React בנויה מהקבצים הבאים (עבור TypeScript):

tsx – הלוגיקה וה-UI של הקומפוננטה (חובה)

css – זה העיצוב של הקומפוננטה (לא חובה)

test.tsx – קובץ בידקות עבור הקומפוננטה - Unit Testing (לא חובה)

ניתן להציג קומפוננטה בתוך index.html

ניתן להציג קומפוננטה בתוך קומפוננטה אחרת.

הקומפוננטה שמכילה נקראת Parent Component

הקומפוננטה המוכלת נקראת Child Component

**React**

זה Framework מבית פייסבוק המאפשרת לבנות אתר SPA מבוסס Components.

ניתן לכתוב React ב-JavaScript או ב-TypeScript (לא כותבים בשניהם יחד).

הדבר הנכון ביותר Ever בעולם כולו הוא – TypeScript.

התקנת פרויקט חדש:

npx create-react-app northwind --template typescript

הרצת הפרויקט: npm start בטרמיל

**Directory Structure**

מבנה הקבצים והתיקיות של מערכת מסוימת.

ב-React נוח לעבוד עם ה-Directory Structure הבא:

src

Components

ProductsArea

AddProduct

AddProduct.css

AddProduct.tsx

UpdateProduct

UpdateProduct.css

UpdateProduct.tsx

ProductList

SuppliersArea

EmployeesArea

Services

Utils

Models

ProductModel.ts

UserModel.ts

**Layout**

זהו הסידור הכללי של האתר.

לכל אתר הגיוני שיהיה סידור כזה. לדוגמה, כותרת בכל הדפים, זכויות יוצרים בתחתית כל דף וכו'...

באתר מבוסס קומפוננטות, ה-Layout הינו Component נפרד שמכיל כל מיני חלקים, שכל אחד מהם הינו Component.

זה אומר שיש אזור נפרד עבור ה-Layout.

קומפוננטת ה-Layout זו הקומפוננטה הראשונה שמוצגת על הדף. כל רכיב אחר מוצג בתוך ה-Layout בצורה ישירה או בתוך רכיבים אחרים.

**CSS Leaking**

זהו מצב בו עיצוב המתבצע בקומפוננטה א' משפיע על קומפוננטה ב'.

קומפוננטה א' מכילה עיצוב של פיסקה אדומה, קומפוננטה ב' מכילה פיסקה שנצבעת באדום.

זהו מצב לא רצוי.

**CLI – Command Line Interface**

אלו קבוצה של פקודות המוכרות ע"י ה-Command Line (Terminal)

**Snippet**

זהו קוד מוכן שניתן לשתול בפקודה אחת

npm i -g react-cli-snippets

זו ספרייה המכילה פקודות CLI עבור Snippets ליצירת Components.

דוגמאות:

create fc HomeArea/Home

create fc ProductsArea/ProductCard --props

create fc ProductsArea/ProductCard -p

**Rendering**

הפעולה של תרגום הקוד שלנו ל-UI הסופי

**Interpolation**

שיבוץ משתנה או ערך בתוך קוד ה-HTML של הקומפוננטה.

**Conditional Rendering**

בדיקת תנאי ולפיו אנו מציגים חלק מה-UI או לא מציגים אותו, או מציגים משהו אחר.

**Displaying Lists**

מתבצעת ע"י קריאה לפונקציה map של מערך או אוסף כלשהו כאשר הפונקציה ממפה כל פריט למשהו אחר. והמשהו האחר שאנו רוצים למפות זהו קטע ה-HTML שברצוננו להציג עבור כל פריט.

כאשר אנו מציגים רשימה, חובה לכל פריט להגדיר מזהה ייחודי ע"י המאפיין key. זאת לצורך Performance.

**Events**

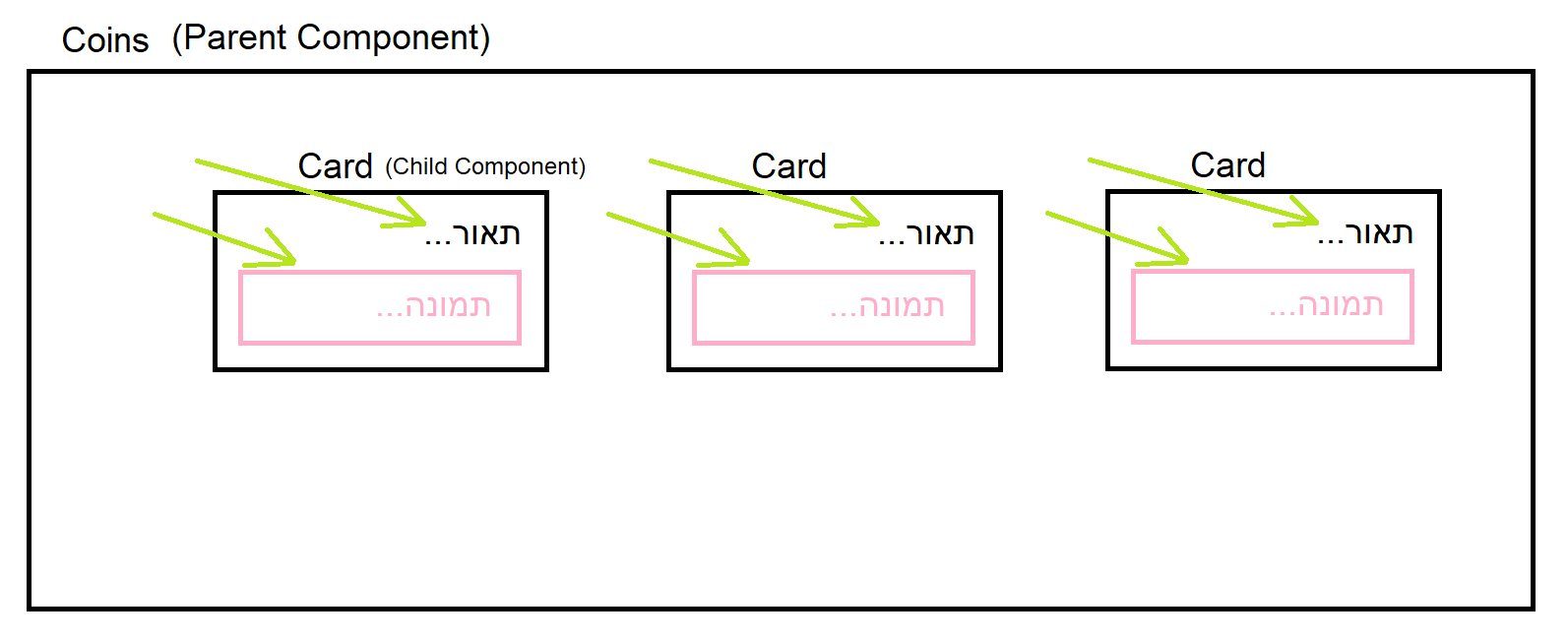
ארוע מתבצע בדומה ל-HTML. שהם הארוע שונה מבחינת Casing (אותיות קטנות / גדולות) מארועי HTML. ארוע נרשם לפונקציה.

**Props**

אובייקט אחד שה-Parent Component יכול לשלוח ל-Child Component.

ה-Child Component מגדיר האם יש לו אובייקט כזה או לא. רק אם הגדיר שיש – ה-Parent יכול לשלוח את ערכי האובייקט.

דוגמה:



**Interface**

מילה שמורה שיוצרת סוג נתונים חדש.

ניתן ב-TypeScript לבנות Literal Object (אובייקט הנבנה ע"י סוגריים מסולסלים {} ) מהסוג הזה.

Class יכולה לרשת interface ואז היא חייבת להגדיר בתוכה את כל מה שיש ב-interface.

**Component State**

מידע הקיים ברמת הקומפוננטה שמתעדכן במהלך חיי הקומפוננטה (בלחיצה על לחצן / בהבאת המידע מהשרת / בטיימר...) ואנו רוצים להציג אותו כחלק מה-UI של הקומפוננטה.

React לא מרנדרת את הקומפוננטה בהפעלת ארועים, בשינוי משתנים רגילים, בקריאה לפונקציות וכו'.

React מרנדרת את הקומפוננטה באחד משני מקרים:

1. מתי שהיא חושבת לנכון (לא ידוע)
2. מתי שאנו משנים State.

לכן אם קיים מידע (משתנה אחד או יותר) שהולך להשתנות במהלך חיי הקומפוננטה ואנו רוצים להציג אותו ברגע שמשתנה – המידע הזה חייב להיות מוגדר כ-State ולא כמשתנה רגיל.

נוהל יצירה ושימוש ב-State שונה בין CC לבין FC.

ב-CC אובייקט ה-State מגיע ממחלקת בסיס בשם Component אותה יורשת מחלקת ה-Component שלנו.

ב-FC ה-State מוגדר ע"י React Hook בשם useState.

**React Hooks**

בגרסאות הראשונות של React היה ניתן לבצע דברים מסוימים רק ע"י CC ולא ע"י FC.

לדוגמה ניהול State.

בגרסה מסוימת, הוסיפו פונקציות מיוחדות שתפקידן לספק עבור FC את אותן יכולות שהיו רק ב-CC.

פונקציות אלו נקראות React Hooks.

מהר מאוד נבנו עוד ועוד פונקציות כאלו שמספקות יכולות נוספות.

דגשים:

* ניתן לקרוא לפונקציה כזו רק ב-FC ולא ב-CC.
* כל פונקציה כזו מתחילה בתחילית use.
* חובה לקרוא לפונקציה כזו ישירות בתוך ה-FC. לא בתוך פונקציות פנימיות ולא בתוך תנאים או לולאות.

**Side-Effect**

אלו פעולות שה-Component מבצע שיוצאות מחוץ לגבולות ה-Component.

לדוגמה, Component גולש לשרת (קריאת AJAX).

לדוגמה, גישה ישירה לתגית DOM כלשהי שלא שייכת ל-Component (שאגב, זה לא מומלץ).

לדוגמה, שימוש ישיר ב-window (פתיחת טיימר)

אסור לבצע Side-Effects ישירות בתוך פונקציית ה-Component! זאת מפני שאנו לא יודעים מתי React בוחרת לרנדר את הקומפוננטה. כל רינדור כזה גורם לה להיות מופעלת מחדש. אם ביצענו Side-Effect ישירות בתוך הקומפוננטה הוא יתבצע מחדש בכל רינדור. לדוגמה – גלישת AJAX – כל רינדור גלישה מחדש לשרת!

ב-CC מבצעים Side-Effects בתוך פונקציות מיוחדות שקיימות במחלקה.

ב-FC מבצעים Side-Effects בתוך React Hook בשם useEffect.

פונקציה זו מסוגלת להריץ Callback כתלות בשינוי מידע.

הפעלת קוד פעם אחת בטעינת ה-FC:

useEffect( () => { … }, [] );

הפעלת קוד בכל פעם שהמשתנים a או b משתנים:

useEffect( () => { … }, [a, b] );

Component מופעל אוטומטית כאשר הוא מוצג.

Component נהרס אוטומטית כאשר הוא כבר לא מוצג יותר על הדף.

קורה שאנו רוצים להפעיל קוד רק כאשר ה-Component נהרס.

לדוגמה, סגירת טיימרים. אם אנו לא סוגרים טיימר, הוא לא יסגר לבד. פעולות סיום, פעולות סגירה...

הפעלת קוד כאשר קומפוננטה נהרסת:

useEffect( () => {

return () => { … };

}, [] );

**Routing**

זהו מנגנון המאפשר להציג Component כתלות ב-Route הנוכחי.

מנגנון זה נכתב בתוך קומפוננטה משלו, לרוב שם Routing. תפקידה להציב את הרכיב המתאים ל-Route הנוכחי.

התקנה: npm i react-router-dom @types/react-router-dom

**Route**

זוהי הכתובת הפנימית של האתר שלנו המוצגת בשורת ה-URL של הדפדפן.

דוגמאות ל-Routes (מסומנים בצהוב):

[http://www.mysite.com**/sports/news**](http://www.mysite.com/sports/news)

[https://some-other-site.co.il**/products/7**](https://some-other-site.co.il/products/7)

[http://cool.com**/employees**](http://cool.com/employees)

[http://www.some-site.com**/**](http://www.some-site.com/)

[http://www.some-site.com](http://www.some-site.com/) (Same as previous)

**AJAX**

גלישה ברקע לאתר מרוחק, ל-WEB API או לשירות כלשהו, החזרת מידע, לרוב לצורך הצגה.

ב-React נהוג לבצע זאת ע"י ספרייה מיועדת לגלישות AJAX בשם Axios.

התקנה: npm i axios

גלישת AJAX נחשב Side-Effect ולכן יש לבצע זאת בתוך useEffect (עבור FC)

התקנת שרת מקומי עבור המידע של Northwind:

npm i -g northwind-back-end

**Model**

זו מחלקת מידע טהור.

לדוגמה: Product, Customer, Supplier, Kitten, Rectangle...

זה לא Component. זה לא UI. זה לא חישובים.

זה רק Data.

מקובל לפעמים לבנות Model ע"י Interface במקום Class.

כאשר אנו מביאים מידע משרת מרוחק, עלינו להגדיר מחלקת Model (או Interface) שמתארים את המידע הזה. כך ב-TypeScript יהיה לנו טיפוס נתונים חזק עבור המידע.

מקובל להגדיר זאת בתיקייה בשם Models המקבילה ל-Components.

מקובל שמחלקה כזו תסתיים בסיומת Model. לדוגמה ProductModel.

**Generics**

טיפוס נתונים המכיל בתוכו סוג שאינו מוגדר באופן ספציפי אלא מי שיוצר אובייקט קובע מהו הסוג.

דוגמה ל-class שאינה Generic:

class Sms {

public message: string; // This is string for ever!

// …

}

---------------------

const sms1 = new Sms();

sms1.message = “Hello”;

sms1.send();

const sms2 = new Sms();

sms2.message = new Image(); // Error!

sms2.send();

פתרון הבעיה ע"י class שהינה Generic:

class Sms<TMessage> {

public message: TMessage;

}

---------------------

const sms1 = new Sms<string>();

sms1.message = “Hello”;

sms1.send();

// sms1.message = 123; // Error

const sms2 = new Sms<Image>();

sms2.message = new Image();

sms2.send();

**Route Parameter**

זהו מידע שאנו כותבים כחלק מה-Route שיכול לקבל ערכים שונים.

דוגמה ל-Route קבוע ללא Route Parameter:

[http://www.mysite.com**/news/sports/football**](http://www.mysite.com/news/sports/football)

דוגמה ל-Route המכיל Route Parameter:

[http://www.mysite.com**/news/article/780**](http://www.mysite.com/news/article/780)

[http://www.mysite.com**/news/article/520**](http://www.mysite.com/news/article/520)

ה-780 או ה-520 או כל מספר שכתוב שם, נחשב Route Parameter שמהווה Value שמתאר דבר מסוים. במקרה הנ"ל מתאר את קוד המאמר.

**Form**

טופס – מאפשר קבלת מידע מהמשתמש.

חשוב מאוד באתרים.

ב-React ניתן לקלוט את המידע מטופס ע"י פונקציית Hook המגיעה מספרייה בשם react-hook-form.

הפונקציה נקראת useForm.

פונקציה זו מאפשרת לקלוט את המידע מהטופס ישירות לתוך אובייקט Model שלנו.

כיום זו הספרייה הכי פופולרית וקלה לקליטת מידע מטופס.

התקנה:

npm i react-hook-form

**שליחת קבצים (תמונות / PDF / וכו')**

לצורך שליחת מידע הכולל גם מאפייני טקסט וגם קבצים, עלינו לשלוח לשרת אובייקט מסוג FormData המכיל את כל המידע כולל הקבצים.

**ולידציה**

אימות קלט המשתמש.

יש שתי דרכים לבצע ולידציה:

1. ע"י validation properties של HTML.

required, min, max, pattern...

<form onSubmit={handleSubmit(submit)}>

                <h2>Add Product</h2>

                <label>Name: </label>

                <input type="text" {...register("name")} required />

                <label>Price: </label>

                <input type="number" step="0.01" {...register("price")} required min="0" max="1000" />

                <label>Stock: </label>

                <input type="number" {...register("stock")} required min="0" max="1000" />

                <label>Image: </label>

                <input type="file" accept="image/\*" {...register("image")} />

                <button>Add</button>

            </form>

1. ולידציה ע"י ספריית useForm המאפשרת להגדיר עבור כל שדה חוקיות ולידציה כולל הודעות שגיאה.

את ההודעות ניתן להציג ע"י formState.errors.\_\_\_\_\_:

<form onSubmit={handleSubmit(submit)}>

                <h2>Add Product</h2>

                <label>Name: </label>

                <input type="text" {...register("name", {

                    required: { value: true, message: "Missing product name" }

                })} />

                <span>{formState.errors.name?.message}</span>

                <label>Price: </label>

                <input type="number" step="0.01" {...register("price", {

                    required: { value: true, message: "Missing price" },

                    min: { value: 0, message: "Price can't be negative" },

                    max: { value: 1000, message: "Price can't exceed 1000" },

                })} />

                <span>{formState.errors.price?.message}</span>

                <label>Stock: </label>

                <input type="number" {...register("stock", {

                    required: { value: true, message: "Missing stock" },

                    min: { value: 0, message: "Stock can't be negative" },

                    max: { value: 1000, message: "Stock can't exceed 1000" },

                })} />

                <span>{formState.errors.stock?.message}</span>

                <label>Image: </label>

                <input type="file" accept="image/\*" {...register("image")} />

                <button>Add</button>

            </form>

**Add / Update / Delete**

**Add:**

אם אין תמונות / קבצים:

const response = await axios.post<ProductModel>(config.productsUrl, product);

אם יש תמונות / קבצים:

const response = await axios.post<ProductModel>(config.productsUrl, formData);

**Update:**

אם אין תמונות / קבצים:

const response = await axios.put<ProductModel>(config.productsUrl + id, product);

אם יש תמונות / קבצים:

const response = await axios.put<ProductModel>(config.productsUrl + id, formData);

**Delete:**

await axios.delete(config.productsUrl + id);

**Lifecycle Hooks**

אלו פונקציות שקיימות רק ב-Class Components שמתבצעות אוטומטית בזמנים ספציפיים בחיי ה-Component.

Dynamic Styling

עיצוב דינאמי של מאפיינים הקיימים ב-State של ה-Component ואז שימוש במאפיינים אלו במאפיין style של כל תגית HTML:

<div className="RandomColor Box" style={{ backgroundColor: this.state.color }}>

**CSS Module**

זוהי טכניקה למניעת CSS Leaking ב-React

בטכניקה זו אנו מגדירים את העיצוב שלנו ב-Class-ים שנמצאים בתוך קובץ בעל סיומת module.css.

ואז מייבאים אובייקט מהקובץ הזה המכיל בתוכו את כל ה-Class-ים

**MUI**

Material UI

זו הספרייה הכי פופולרית לעיצוב Components ב-React.

אלו Components מוכנים לשימוש עבור רכיבי HTML בסיסיים.

לדוגמה במקום להשתמש ב-<input type=”text /> משתמשים בקומפוננטה שמציגה תיבת טקסט בצורה מעוצבת.

התקנה:

ליבת הספרייה: npm i @material-ui/core

אייקונים: npm i @material-ui/icons

**Redux**

ארכיטקטורה לשמירת מידע במקום אחד עבור כל האפליקציה (צד לקוח).

אם ישנם מספר Components שצריכים את אותו המידע, נכון יהיה לשמור את המידע במקום מרוכז אחד ולא רק באחד מה-Components.

**Service**

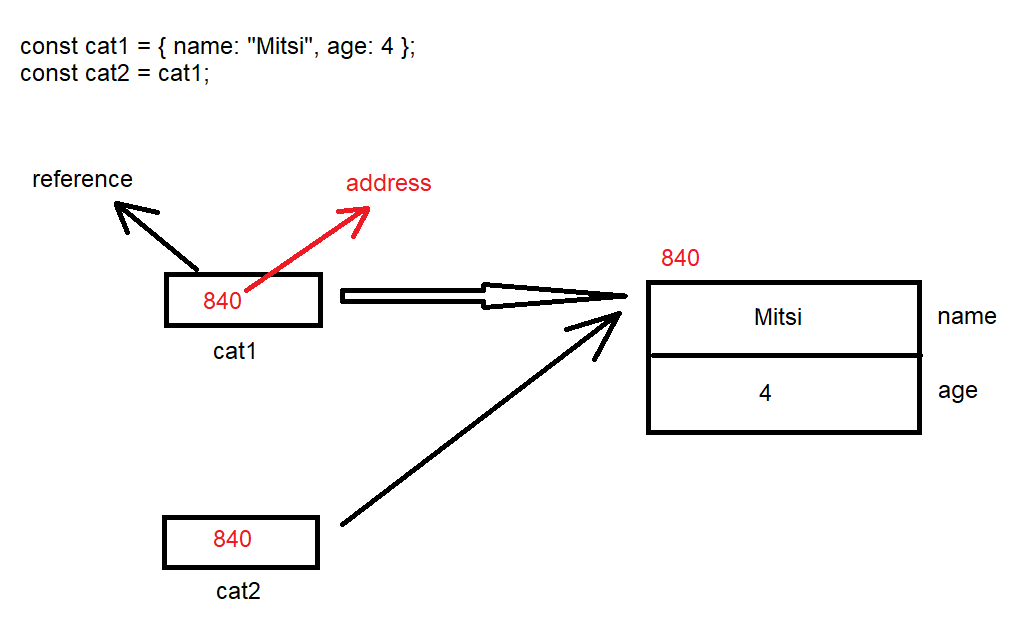
מחלקת לוגיקה בלבד (ב-Frontend). אין בה UI. היא מספקת לוגיקה בלבד ל-Components או ל-Services אחרים.

לוגיקה צריכה להיות ב-Service מתאים, כאשר ה-Components משתמשים עם ה-Service לביצוע הלוגיקה.

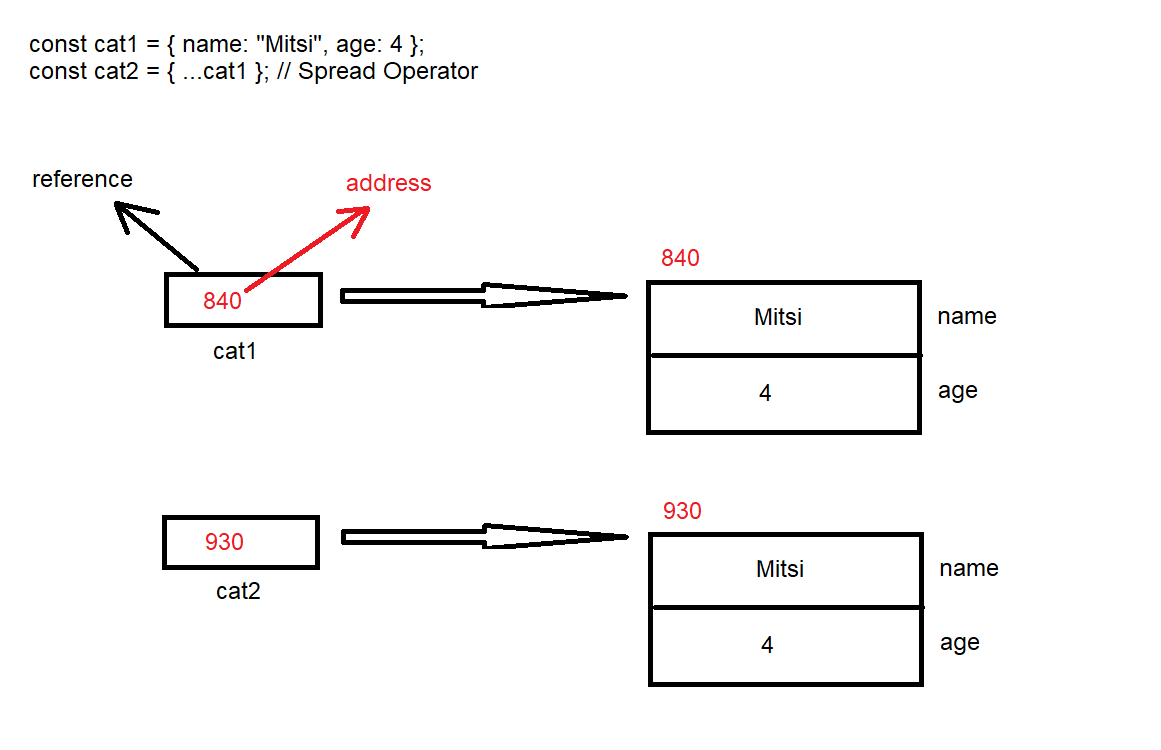
Services מחולקים לפי נושאים / סוגי לוגיקות וכו'. לדוגמה לוגיקה הקשורה למוצרים תהיה ב-ProductsService, לוגיקה הקשורה לעובדים תהיה ב-EmployeesService.

תזכורת לגבי Spread Operator:

ללא שימוש ב-Spread Operator:



עם שימוש ב-Spread Operator:



**Auth**

זהו קיצור של שני מונחים:

Authentication: אימות – לדעת מי זה המשתמש במערכת. ניתן לבצע רק אם המשתמש ביצע הרשמה (Register) או כניסה (Login).

Authorization: הרשאות – מה המשתמש יכול לבצע במערכת. יכול להיות משתמש אנונימי (לא יודעים מי הוא), יכול להיות משתמש שביצע Login – ואז ניתן לדעת מי הוא ומה הוא יכול לבצע.

נוהל שילוב Auth ב-React:

1. בניית Model המייצג משתמש (UserModel).
2. בניית AuthState ב-Redux המכיל את אובייקט המשתמש שהשרת יחזיר, ומאפשר ביצוע שלושה דברים:
   * Register
   * Login
   * Logout

יש להתקין את npm i jwt-decode עבור פענוח מחרוזת ה-token.

1. שילוב ה-authReducer ב-store של Redux.
2. בניית פונקציית register ב-AuthService לתקשורת עם השרת (יש להגדיר לשרת את הנתיבים ב-config).
3. בניית טופס עבור הרשמה – ב-Component נפרד, כולל Route המציג Component זה.
4. בניית Model המייצג פרטי Login (CredentialsModel).
5. בניית פונקציית login ב-AuthService לתקשורת עם השרת (יש להגדיר לשרת את הנתיבים ב-config).
6. בניית טופס עבור כניסה – ב-Component נפרד, כולל Route המציג Component זה.
7. בניית פונקציית logout ב-AuthService לביצוע logout.
8. בניית רכיב Logout נפרד, כולל Route הגולש ל-Component זה.
9. בניית תפריט המאפשר לבצע
   * Register | Login

או

* + Logout

אפשר להציג באזור הזה גם את פרטי המשתמש כמו שם מלא.

יש לבצע subscribe בכדי לקבל שינויים שהתבצעו ב-Store בכדי לעדכן את התפריט.

יש לבצע unsubscribe בהריסת הרכיב, בכדי להפסיק לקבל שינויים .

1. בניית Interceptor עבור axios השולח את ה-Token לשרת (אם קיים) בכל Request.

**Interceptor**

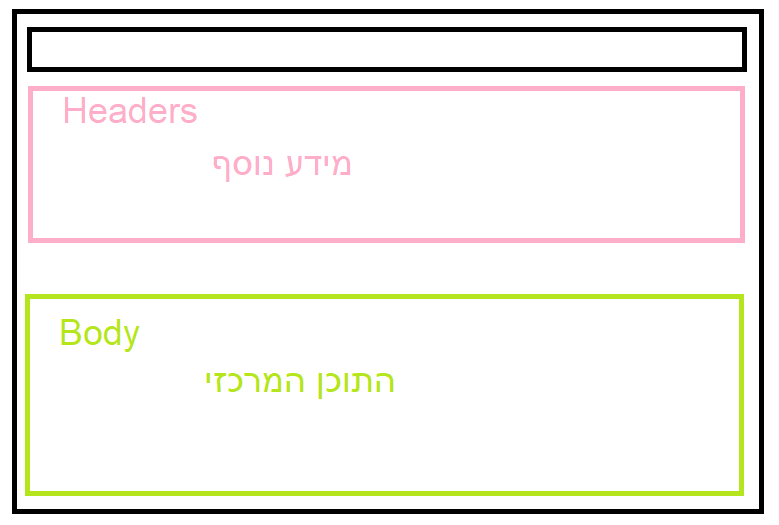
מנגנון הנכנס לפעולה בכל Request או Response בצורה אוטומטית ויכול לשנות את ה-Request או את ה-Response.

**Request Headers**

אזור ב-Request המכיל מידע נוסף שנשלח מה-Frontend ל-Backend.

**Response Headers**

אזור ב-Response המכיל מידע נוסף שנשלח מה-Backend ל-Frontend.



**הצגת הודעות שגיאה**

שגיאות יכולות להתרחש עקב קוד ב-Front שמעלה חריגה ועקב קוד ב-Back שמחזיר שגיאה.

אין מוסכמה עדיין בעולם לגבי הפורמט של אובייקט השגיאה. כלומר, שרתים יכולים להחזיר כל אובייקט שהוא לייצוג שגיאה.

כן יש קוד ססטוס לייצוג שגיאות שהשרת מחזיר ללקוח:

400 – שגיאת ולידציה או מידע לא חוקי שנשלח

401 / 403 – שגיאת Authorization / Authentication

404 – דף או כתובת שאינם קיימים

500 – קריסה בצד השרת

בטיפול בשגיאות בצד הלקוח עלינו לבצע גם:

* הצגת ההודעה הנכונה של השגיאה
* לא כדאי להציג alert-ים. alert תוקע את זרימת התוכנית, לא הכי ברור, לא ניתן לעיצוב...

קיימות מגוון ספריות להצגת הודעות. לדוגמה: npm i notyf

**העלאה ל-Production**

העלאת האתר לשרת אמיתי הקיים אונליין:

אפשרויות:

1. מחשב שרת שלכם הניתן לגישה ע"י RDP
2. חברת אחסון הניתן לגישה ע"י ממשק Web-י. האחסון לרוב מכיל תיקייה אחת אליה מעלים את הקבצים שמייצגת אתר אחד ויחיד.
3. ענן – Cloud – מערכת המאפשרת להעלות קבצים לפרויקט המכיל תשתית לפי הדרישה שלנו.
   * Azure
   * AWS
   * Google Cloud
   * Firebase
   * Heroku
   * ועוד