Angular 6

|  |
| --- |
| הגדרות  אנגולר זה פריימוורק של JS (מערכת מבוססת JS) המשמשת לבניית אפליקציות בצד הקליינט (בדפדפן).  אנגולר משתמשת ב HTML, CSS, Js  אנגולר מוסיפה ל HTML תגיות של IF וקונדישנים נוספים, משתנים לוקאליים וכו'  יש לה יכולות DATA BIND מעולות – אפשר להציג שדות מ אובייקטים של דאדא, לעקוב אחר שינויים ולטפל בעדכונים שהיוזר עושה לדאטא.  יש לה יכולות לפרק את האפליקציה לחלקים (מודולים) שמאפשרים קוד REUSE  יש לה יכולות מובנות של תקשורת עם השרת (לצורך קבלת מידע למשל)  יתרונות נוספים:  מהירה – אתחול מהיר, מעקב מהיר אחרי שינויים, מרנדרת מהר.  מודרנית – משתמשת בפיצרים חדשים שלcli JS כמו מח', מודולים וכו'  יש לה API פשוט ומעט דברים BUILD IN שצריך ללמוד  אנגולר היא מודולרית – בנויה ממודולים רבים. מודולים לדוגמה: core, http, router .    \*בקובץ packaje.json ניתן לראות את כל המודולים של אנגולר שהם חלק מהאפליקציה שלי.    אנגולר מוגדרת בתור SPA – Single Page Application  כלומר – היא מכילה קובץ HTML יחיד, אשר מחליפים בו חלקים של ה HTML. |
| אנטומיה של אפליקציית אנגולר  אפליקציית אנגולר בנויה ממודולים אשר מכילים קומפוננטות , סרביסים (שמספקים פונקציונליות לרוחב קומפוננטות). |
| מבנה אפליקציית אנגולר:  כל קבצי המקור נמצאים תחת פולדר בשם src (זו הקונבנציה)  מתחתיו – פולדר app. מכיל את כל קבצי המקור שמכילים את הקוד הספציפי לאפליקציה שלי.  את כל הקוד שלי אשים בפולדרים (הקוד מחולק לחלקים)  בנוסף – יש את כל הקבצים של תבנית אפליקציית אנגולר- קבצי קונפיגורציה ו קבצי SETUP.   * מכיל בין היתר את קובץ package.json שמגדיר את כל הספריות שהאפליקציה תלויה בהם. |
| TypeScript  שפה שמתורגמת ל JS.  יתרונותיה:  היא Strong Typed - כלומר לכל דבר מוגדר סוג נתונים:  לכל פרופרטי  לכל פרמטר של פונקציה  לכל פונקציה מוגדר סוג הנתונים שהיא מחזירה  זה עוזר להקטין את כמות השגיאות על ידי בדיקת סינטקס שהעורך מבצע  יש לה דוקומנטציה מובנית  יש לה בדיקת סינטקס  יש לה ניווט בקוד  יש לה יכולות ריפקטורינג  איך TS יודעת את סוג הנתונים כשמשתמשים בספריות JS שאינן strong typed? על ידי שימוש בקבצי הגדרות סוגי נתונים של TypeScript (קבצים עם סיומת XXX.d.ts (XXX=שם הספריה)). הקבצים האלה מכילים את הגדרות סוגי הנתונים עבור כל דבר בספרייה.  TypeScript מכילה מחלקות, אינטרפייסים, ירושה  סביבה שאפשר להתנסות בה ב TypeScript    VisualStudio תומכת ב TypeScript. VS מספקת ל typescript - intellisence, בדיקת סינטקס, וריפקטורינג.  JS - רקע  התקן של שפת JS נקרא ECMAScript (ES). ES3 נתמכת על ידי בראוזרים ישנים. ES5 נתמכת על ידי רוב הדפדפנים היום.  ES6 נקראת ES 2015. רוב הדפדפנים עוד לא תומכים ב ES 2015 – לכן הוא צריך לעבור תרגום (transpile) – כלומר צריך כלי שמקמפל – כלומר ממיר את הסינטקס של ES 2015 לסינטקס של ES5 לפני שהבראוזר מבצע את הקוד.  ככה – המתכנת יכול להינות מ ES 2015 והדפדפנים יכולים להבין את הקוד. |
| Npm  תוכנת עזר של command line (אין לה ממשק גרפי) שיודעת לדבר עם מאגר של פרוייקטים open source , והיא מאוד יעילה עבור ספריות JS. משמשת להתקנה של ספריות, packages ואפליקציות.  צריך NPM כדי להתקין את ספריות אנגולר  כדי להתקין NPM צריך להתקין NODE (NPM מגיע עם NODE.JS)  package.json  נדמה לי שזה קובץ שמשמש את NPM. הקובץ מגדיר את כל הספריות JS שהאפליקציה תלויה בהם, וגם CSS וכל ספריה חיצונית שצריך להוריד כדי להריץ את הפרויקט. (הגדרות אלו ב: dependencies של הקובץ)  יש בו גם חלק scripts שמגדיר פקודות עבור NPM ומה משמעותן (איזה פקודות להריץ כשהוא מקבל פקודה מסוימת)    כדי להתקין את הספריות שמקונפגות ב package.json צריך לנווט לפולדר הזה ב CMD ולהריץ את הפקודה npm install. Npm יתקין את כל הספריות בפולדר node\_modules , וכל ספריה עם ה dependencies שלה. הפולדר node\_modules הוא גדול, אז לא כדאי לצרף אותו ל version control כמו GIT.  כדי להריץ את הפקודות- נקליד ב CMD - למשל npm start - זה יגרום ל NPM להריץ את הפקודה ng serve -o.  הפקודה: ng תריץ את ה angular cle. הפקודה serve בונה את האפליקציה ומתחילה שרת אינטרנט (השרת פיתוח של angular cli) |
|  |
|  |

XX

|  |
| --- |
| הסבר כללי על האתחול (bootstrap) של אפליקציית אנגולר  באנגולר האפליקציה מתחילה כך: מגדירים את המודול הראשי של האפליקציה, ובמודול הראשי מגדירים את הקומפו' הראשית. אומרים למודול הראשי איזו קומפו' לאתחל כשהאפליקציה עולה (bootstrap). בנוסף – בקובץ ה index.html צריך להוסיף את הקומפו' הראשית.  בנוסף, כל אפליקציה מתחילה מקובץ מסוים. שלי מתחילה מ main.ts. זה הקובץ הראשון שאנגולר קוראת כשהיא מתחילה את האפילקציה שלי. יש בו רק שורת קוד אחת שמאתחלת את המודול הראשי של האפליקציה: bootstrap(AppModule);  יש לקנפג את האפליקציה להתחיל כך:   1. בקובץ index.html – נוסיף את הקומפו' הראשית AppComponent. נעשה את זה על ידי הוספת דיירקטיב עבור הקומפו' הראשית. 2. במודול הראשי של האפליקציה נייבא את הקומפו' הראשית ונגדיר אותה כקומפו' שעולה בזמן ה bootstrap 3. מאיזה קובץ האפליקציה מתחילה? מקובץ main.ts – שם צריך להוסיף את שורת הקוד שעושה bootstrap עם המודול הראשי.   דוגמה ל index.html. באדום – אלמנט מיוחד app-root שלנו (לא שייך ל HTML) שמייצג את הקומפו' הראשית    דוגמה למודול הראשי של האפליקציה AppModule. בדקורייטור שלו מופיעה הקונפיגורציה של ה bootstrap  דוגמה של המודול: בכחול ההכרזה על הקומפו', באדום – הגדרת ה bootstrap של המודול    דוגמה לקומפו' הראשית AppComponent . בכחול – הגדרת שם הדיירקטיב כ app-root – זה שם האלמנט המיוחד שהוספתי לHTML בקובץ index.html    דוגמה לשורת הקוד בקובץ main.ts : (באדום)    \*הגדרה - בוטסטראפ = פרוסס (תוכנה) שטוען את עצמו וגורם לעצמו להתחיל לעבוד. |

XX

|  |
| --- |
| Angular CLI  כלי של command line (אין לו ממשק גרפי) שעוזר ליצור אפליקציות אנגולר:   1. מייצר אפליקציה חדשה - מייצר את קבצי ה SETUP ותבנית ריקה עבור אפליקציית אנגולר. 2. משמש ליצירת קבצים (שלד ריק) של קומפו', מודולים, סרביסים ועוד – בזמן כתיבת האפליקציה עצמה 3. משמש להוספת קוד-שלד לבדיקות של האפליקציה (unit וגם end to end) 4. מבצע בדיקות עבור האפליקציה. 5. מכין את האפליקציה ל deployment- עושה מיניפיקציה ואורז את הקוד בקבצים השונים לקובץ יחיד   ה CLI נמצא כאן:    התקנת ה CLI  \*זו התקנה גלובלית כדי שנוכל להתמש בו מכל פולדר שהוא  Npm install -g @angular/cli  איך לבדוק מה גרסת ה CLI שלי?  ng v |
| עזרה  כדי לראות את הפקודות השונות של ה CLI נקליד את הפקודה:  ng help  כדי לקבל עזרה על פקודה יחידה:  דוגמה: ירוק- פקודת העזרה, אדום- הסינטקס של הפקודה, כחול- מטרת הפקודה, ורוד- שמות נרדפים של הפקודה (aliases), תכלת- רשימת ה flags של הפקודה  ng XX (שם פקודה) –help |
| ng new  Getting Started – כך מייצרים אפליקצית אנגולר חדשה  ng new APP\_NAME  כך מייצרים אפליקציית אנגולר חדשה עם מודול של ראוטינג כבר מוכן:  ng new XXX --routing (נדמה לי)  ה CLI מייצר מבנה ספריות עם קבצים עבור האפליקציה, ובנוסף יוצר קובץ package.json. הוא אוטו' מריץ את npm install אשר מתקינה את כל הספריות שמקונפגות ב package.json  דוגמה: אדום- פקודת ה new, כחול- רשימת הפולדרים והקבצים שה CLI יצר, ורוד- הודעה שכרגע npm מתקין את הספריות שמקונפגות ב package.json    בסוף תהליך ההקמה- נקבל הודעת success :  דוגמה: אדום- npm סיים להתקין את הספריות מקובץ packages.json, כחול- ..., ורוד- האפליקציה החדשה נוצרה בהצלחה |
| ng serve  כדי להריץ את האפליקציה בשרת הפיתוח של ה CLI נריץ את הפקודה ng serve.   1. ה CLI יתחיל שרת פיתוח המאזין על פורט 4200 2. בונה את האפליקציה- מקמפל את הקוד ואורז אותו בקבצי JS ו js.map (קובץ map משמש את הדפדפן כך שיוכל להראות לי את קבצי הקוד כמו שהם כתובים בסביבת הפיתוח שלי, ולא minified וכו'.) 3. מוסיף תגיות script מתאימות לקובץ index.html – כדי שיכלול את הקוד.   \*הקומפיילר עושה WATCH על הקבצים, וכשיש שינויים – הוא מקמפל אוטו' מחדש.  דוגמה: אדום- פקודת ng serve, כחול- הודעה שהשרת פיתוח עובד ומאזין על פורט 4200, ורוד- הקבצים שהקומפיילר ייצר לנו, ירוק- הודע שהאפליקציה קומפלה בהצלחה  כעת ניתן לגלוש אל האפליקציה בבראוזר    \*הקבצי JS של האפליקציה לא נשמרים בדיסק, ה CLI מייצר אותם ושומר אותם בזיכרון ומגיש אותם מהזיכרון, לא שומר אותם על הדיסק.  \*כל הקוד שלי מקומפל לתוך הקובץ (שנקרא bundle) main.bundle.js  דוגמה: ניתן לראות את קבצי ה JS של האפליקציה שלי בבראוזר    \*ה CLI ייצר גם קבצי map- לכן ניתן לראות תחת webpack בבראוזר את קבצי המקור ב typescript.  דוגמה: אדום- באזור ה webpack , כחול- כל קבצי המקור של האפליקציה כולל הנתיב שלהם בדיסק |
| ng build  בונה את האפליקציה ומכין deployment   1. מקמפל ואורז את הקוד בקבצים (bundles) 2. מייצר עותק של קובץ index.html עם תגיות סקריפט מתאימות 3. מייצר פולדר dist ושם שם את הקבצים   ng build –prod  כשה CLI מקמפל לפרודקשן – הוא עושה את הדברים הבאים:   1. הוא עושה minify וגם uglify וגם tree shaking – תהליך ש"מנער" את העץ שלנו וזורק קוד שלא בשימוש 2. מוסיף קוד HASH לשם של כל קובץ. זה למטרה של מניעת caching בבראוזר. 3. לא מייצר קבצי map 4. ה CLI משתמש ב AOT קומפיילר ועושה pre-compile לטמפלייטים.   זה מאוד משפר ביצועים כי זה מקטין מאוד את הקובץ vendor.js וגם משפר את מהירות האפליקציה  הסבר: aot vs jit  aot – קומפיילר שמקמפל לפני הזמן – Ahead of Time Compiler  JIT - Just In Time compilation  JIT - אם מקמפלים JIT זה אומר שהקומפיילר ירוץ בדפדפן בזמן טעינת האפליקציה. זה מאיט את האפליקציה כי היוזר צריך לחכות לקומפילציה. וגם- הקליינט צריך את הקומפיילר של אנגולר, לכן אנגולר מוסיף את קוד הקומפיילר לחבילת ה vendor.js וזה מגדיל אותה מאוד (כמעט מגה נוסף נדמה לי). זה גם לא יעיל כי האפליקציה היא אותה אפליקציה לכל הדפדפנים, וכל דפדפן צריך לקמפל בעצמו את אותו דבר. עדיף לקמפל מראש לכולם.  אם מקמפלים AOT - האפליקציה יותר מהירה (כי הקוד כבר מגיע מקומפל) ויותר קטנה (כי הדפדפן לא צריך את הקומפיילר של אנגולר, לכן לא מוסיפים אותו לחבילת ה vendor.js - הבדל של כמעט מגה)  הסבר על תהליך הקומפילציה של אנגולר  עבור כל קומפו הקומפיילר מייצר קוד JS אשר מייצר את האלמנטים של ה DOM (את האובייקטים) ומוסיף אותם ל DOM.  לדוגמה עבור טמפלייט עם div שבתוכו h1 - אנגולר ייצר בתהליך הקומפילציה קוד דומה לזה:  var div = document.createElement('div');  var h1 = document.createElement('h1');  h1.innerHtml = 'hello world';  div.appendChild(h1);  נדמה לי שמרגע זה אין ל HTML של הטמפלייט יותר תפקיד. כלומר- אנגולר לא מוסיף את ה HTML לקובץ index אלא מייצר לפיו קוד JS שמוסיף אלמנטים ל DOM, ואז הדפדפן מעדכן את התצוגה מתוך ה DOM. הדפדפן לא משתמש ב HTML של הטמפלייט. נדמה לי שה HTML הזה הולך לפח אחרי תהליך הקומפליציה.   1. אופציות נוספות: אפשר לשנות את הנתיב של האפליקציה (ב URL בעזרת הפלאג –base href, לבחור סביבת דיפלויימנט בעזרת -prod |
| ng lint  כדי שהקוד יהיה קריא וקל לתחזוקה כדאי להשתמש בלינטר linter. לינטר זו תוכנה שמבצעת בדיקה של הקוד כדי לראות אם הוא מפר חוקים.  כך ניתן לוודא שהקוד אחיד- למשל תמיד משתמשים ב גרש יחיד במקום בגרשיים - על סטרינגים.  יש לינטר פופולרי של typescript שנקרא tslint. הוא נמצא ב github, ואנגולר CLI מוסיף אותו אוטו' לכל אפליקציה חדשה (הוא מוסיף אותו לקובץ package.json ואחר כך מריץ npm install וזה מותקן אוטו')  הפקודה ng lint תציג לנו בטרמינל רשימת שגיאות שאפשר לתקן ידנית.  ng lint --fix - יתקן אוטו' את הדברים שקל לתקן, ויציג הודעות רק על השגיאות שהוא לא יכול לתקן אוטו'.  איך לקנפג את ה tslint  קובץ הקונפיגורציה של ה tsling נקרא tsling.json והוא נמצא בפולדר הראשי של האפליקציה.  כל החוקים מוגדרים בקובץ הזה וניתן לקנפג אותם לפי הקונבנציות של צוות הפיתוח.  דוגמה לחוק הגרשיים: אדום- שם החוק quotemark. כדי לכבות אותו משנים את true הכחול ל false. תכלת- מוגדר כגרש יחיד. כדי לשנות לגרשיים צריך לשנות את מ single ל double.    חוקים נפוצים:  no-trailing-whitespace - מתריע כאשר יש רווחים בסוף שורה |

XX

|  |
| --- |
| VSCode  לחיצה על הנורה שבצד שמאל  ctrl + נקודה    אם קיצורים לא עובדים לי - צריך להסתכל על ה view menu שבראש המסך, אפשר לראות שם את הקיצורים. |
| extentions חשובים  בצד שמאל לוחצים על האייקון של ה extentions. דוגמה:    tslint  \*בכל מקרה כדאי לעשות ng lint לפני שמעלים את הקוד ל source.  צריך לעשות Install ואז reload    מתריע בזמן כתיבת הקוד על שגיאות של tslint.  דוגמה- רווחים מיותרים בסוף שורה:    דוגמה- גרשיים כפולים במקום יחיד. אם נלחץ על הנורה (או בקיצור ctrl + נקודה) תיפתח התיבה התכלת עם האפציות שמציעה לתקן לגרש יחיד, לעשות disable לחוק הזה של tslint, או נדמה לי לפתוח את קובץ הקונפיגורציה של tslint.    איך לתקן את כל השגיאות בדף בבת אחת:  לפתוח את שורת הפקודה (shift+ctrl+p) ולכתוב את הפקודה:  tslint  ולבחור באופציה fix all auto fixable problems |
|  |
|  |

XX

|  |
| --- |
| קומפוננטה  קומפו משמשת לבניית ממשק המשתמש (התצוגה)  קומפו' יכולות לקנן אחת בתוך השניה, כך שאפשר לחלק את ה UI לחלקים רבים שאפשר לעשות להם REUSE או שיש להם תפקידים שונים וצריך להפריד אותם  בקומפו משתמשים גם ב data binding ובדיירקטיבים.  כל קומפו שייכת למודול. קומפו' יכולה להיות שייכת רק למודול אחד.  כדי ליצור קומפו:  0. ניצור פולדר חדש בשביל הקומפו' (זה נהוג) ונקרא לו בשם הקומפו (מופרד עם מקפים, לא camelCase).  1. ניצור טמפלייט - ניצור קובץ HTML חדש ונקרא לו XXX.component.html  2. ניצור קוד עבור הטמפלייט - ניצור קובץ חדש XXX.component.ts, נכתוב בו מח' חדשה XxxComponent ונעשה לה export  2. נוסיף דקורייטור @Component() למח' החדשה שיצרנו XxxComponent  4. נגדיר בפרופרטי TemplateUrl של הדקורייטור @Component - את הטמפלייט שיצרנו ב 1  5. אופציונלי- ניתן להוסיף css styles בקובץ נפרד ולהוסיף אותו לקומפו  דוגמה:  קומפו' בשם ProductListComponent  ורוד פולדר בשם product-list עבור הקומפו. אדום קובץ טמפלייט מסוג html. כחול קובץ עבור המח' מסוג ts. שם הקומפו מופרד עם מקפים ולא עם camelCase.    קובץ המח'. תכלת- המח'. שם המח' ב camelCase. צהוב- נוסיף export כדי שאפשר יהיה לגשת אל המח' ממודולים (קבצים) אחרים. אדום- נוסיף metadata עם הדקורייטור @Component() שהוספנו למח'. ורוד- נייבא את הדקורייטור Comopnent מתוך מודול הבסיס של אנגולר (angular core). כחול- ניתן שם לדיירקטיב. ירוק- נגדיר את הטמפלייט של הקומפו' כקובץ HTML נפרד (התוים "./" מסמנים את הפולדר הנוכחי)    נכריז על הקומפו' במודול הראשי של האפליקציה. כחול- המודול הראשי (נקרא AppModule). אדום- נייבא את הקומפו (המח') מהמודול שבו היא מוכרזת (product-list.component.ts). ורוד- נכריז עליה ב Declerations של המודול AppModule.    נשתמש בקומפו' כדיירקטיב (כלומר נוסיף את התגית לתוך טמפלייט אחר). דיירקטיב=תגיות שאינן שייכות ל HTML, ואשר מעשירות את ה HTML. למעשה יצרנו פה תגית שמעשירה את ה HTML ב HTML נוסף (עם קוד מקושר אליה) . נוסיף דיירקטיב (שם הדיירקטיב- השם שהוגדר בסלקטור ב metadata של הקומפו'). אדום- הדיירקטיב החדש של הקומפו החדשה שיצרנו, בתוך קומפו אבא. |
| קומוננטה - המשך  קומפו בנויה מ:   * טמפלייט (המגדירה תצוגה VIEW) * מחלקה (קוד), * מידע נוסף (metadata) – שמוגדר בדקורייטור בשם @Component   כלומר- קומפו מורכבת מתצוגה (מוגדרת בטמפלייט), מקוד (מוגדר במחלקה), ומידע פנימי (מוגדר בדקורייטור)  טמפלייט – מורכבת מ HTML בתוספת דיירקטיבים ו data binding של אנגולר  מחלקה – קוד שמקושר ל VIEW. מורכבת מ TypeScript. המח' מכילה פרופרטיס (מידע לשימוש ב VIEW) , ומתודות (לביצוע פעולות עבור ה VIEW)  מידע נוסף (metadata) – מספק מידע על הקומפו' עבור אנגולר, באמצעות DECORATOR בשם @Component()    \*באנגולר מקובל לקרוא למח' עם סיומת המילה Component. לדוגמה- ProductComponent. אחר כך משתמשים בשם המח' בתור שם הקומפו' (המשמש להוספת הקומפו' בקוד)  \*באנגולר מקובל לקרוא לקומפו' הראשית של האפליקציה בשם AppComponent  \*באנגולר מקובל לקרוא למח' עם אות גדולה, ולפרופרטיס עם אות קטנה.  \*דקורטור זה פיצ'ר של JS - זו פונקציה אשר מוסיפה מידע metadata (מידע על האובייקט אותו היא מקשטת) . משתמשים בדקורטור על מחלקות, ממברים של מחלקות (פרופרטיס ומתודות), ועל פרמטרים של פונקציות.  דקורטור תמיד מתחיל בתו @. למשל- @Component() – הדקורטור של קומפוננטות  באנגולר יש כמה דקורטורים מובנים בתוך הפריימוורק - @Component למשל.  \*הסבר על הדקורייטור @Component(): (הסבר כללי על דקורייטורים בהמשך)  הפונקציה הזו מקבלת אובייקט כפרמטר, ולאובייקט יש הרבה פרופרטיס:  selector - כאשר יש selector לקומפו- אפשר להשתמש בה כדיירקטיב. שם הדיירקטיב יהיה שם הסלקטור (דיירקטיב ב HTML במקום אחר באפליקציה)  template – חובה. קומפו' חייבת תצוגה, אחרת היא לא קומפו'. אם הקומפו בקובץ נפרד - משתמשים ב templateUrl במקום  inline vs linked templates  כאשר הטמפלייט מוגדרת בקובץ HTML נפרד - היא נקראת linked template  כאשר היא מוגדרת כסטרינג inline בתוך הדקורייטור - היא נקראת inline template)  שמות של קבצי הקומפו  בקונבנציה - קוראים לקובץ הטמפלייט בשם של הקומפו + המילה קומפו + סיומת .html  קוראים לקובץ הקומפו בשם של הקומפו + המילה קומפו + סיומת ts  דוגמה: לקומפו' product יהיו הקבצים הבאים:  product.component.ts  product.component.html  איך אנגולר יודעת לפרש את הדיירקטיב (סלקטור)? לדוגמה- אם נשים דיירקטיב <bbb> בקומפו AAA איך אנגולר תדע לפרש את הדיירקטיב הזה?  כשהיא רואה דיירקטיב שהיא לא מכירה בקומפו AAA- היא הולכת למודול שבו מתגוררת קומפו AAA ומחפשת בהכרזות שלו אחר קומפו XXX שב metadata שלה מוגדר סלקטור בשם bbb. אם היא לא מוצאת- היא תחפש באותו אופן במודולים בנים (מודולים שמיובאים לתוך המודול האב).  הדיירקטיב הזה חייב להיות מוכרז (declared) במודול הזה או במודולים בנים שלו. אחרת נחטוף שגיאה.  דוגמה לשגיאה: כחול pm-products אינה אלמנט מוכר. Pm-product הוא קומפו שלא הכריזו עליו במודול. |
| קומפוננטה - המשך |
| דקורטור Decorator  כמו אטריביוטים ב C# - ב JS משתמשים בדקורטורים כדי להוסיף metadata על אובייקטים.  משתמשים בדקורטור על מחלקות, ממברים של מחלקות (פרופרטיס ומתודות), ועל פרמטרים של פונקציות.  דקורטור תמיד מתחיל בתו @ (נדמה לי שזה ככה, אבל אם לא תמיד – אז תמיד באנגולר)  דקורטור זו פונקציית JS אשר מוסיפה מידע metadata (מידע על האובייקט אותו היא מקשטת) .  הסקופ של הדקורטור הוא האובייקט אותו היא מקשטת.  בין הדקורייטור לאובייקט שאותו היא מקשטת לא שמים נקודה פסיק ;  דוגמה לדקורטור (משתמשים בדקורטור הזה באנגולר, שזו פריימוורק של JS : זהו דקורטור שמקשטת מחלקה. |
| הוספת style לקומפו  אם רוצים להוסיף style ספציפי לקומפו:  נגדיר קובץ סטייל XXX.component.css ונגדיר אותו בפרופרטי styleUrl של הדקורייטור @Component(). (הפרופרטי מקבל מערך של סטרינגים שהם שמות קבצים)  דוגמה:    הסטיילים שמוגדרים בקובץ הזה זמינים רק בקומפו' הזאת, לא ברחבי האפליקציה |
| lifecycle hooks  לקומפו יש מחזור חיים שאנגולר מנהלת:  אנגולר יוצרת את הקומפו',  מרנדרת אותה,  יוצרת ומרנדרת את הילדים שלה,  מבצעת עיבוד של השינויים אם קורים שינויים בפרופרטיס של code behind כתוצאה מ two way data binding,  ומחסלת אותה לפני שמסירה את הטמפלייט מה DOM.  lifecycle hooks מספקים דרך לנצל את השלבים האלה כדי לבצע פעולות שאנחנו צריכים לתפקוד של האפליקציה. לדוגמה: טעינת מידע מהשרת בזמן יצירת הקומפו.  הרצף של ה lifecycle hooks:  OnChange() - OnInit() - DoCheck() - AfterContentInit() - AfterContentChecked() - AfterViewInit() - AfterViewChecked() - OnDestroy().  3 ה lifecycle hooks השימושיים ביותר:  OnInit - מתבצע אחרי שאנגולר איתחלה את הפרופרטיס של databind (של המח'). משמש להשגת מידע מהשרת, ולאתחול הקומפו (השמת ערכים בפרופרטיס וכו')  \*אם רוצים לאתחל משתנים- צריך לעשות זאת בקונסטרקטור, לא ב OnInit, כי ב- OnInit הם כבר מאותחלים.  OnChanges - מתבצע אחרי שינוי בפרופרטיס מסוג input. (נדמה לי שרלוונטי רק במקרה של קומפו' מקוננות בתוך קומפו אבא)  OnDestroy - מתבצע לפני שאנגולר מחסלת את הקומפו. משמש לנקיון וחיסול דברים.  יש באנגולר אינטרפייס מובנה לכל lifecycle hook. כדי לנצל את ה hook צריך לממש את האינטרפייס (זה יעבוד גם בלי האינטרפייס, פשוט עם הכרזה על מתודה בשם הנכון. זה בגלל שאינטרפייסים לא מתורגמים ל JS (ראה פירוט אינטרפייסים בחלק TypeScript) אבל זה יותר בטיחותי בזמן כתיבת הקוד ודיבאג לכתוב את זה בעזרת מימוש של אינטרפייס)  כל אינטרפייס כזה מגדיר מתודה יחידה ששמה הוא כשם ה hook עם קידומת ng.  דוגמה לשימוש ב OnInit: אדום- האינטרפייס, תכלת- המתודה    דוגמה לשימוש ב lifecycle hook - OnChanges. אדום- מימוש האינטרפייס. כחול- מימוש הפונקציה ngOnChanges |
| filter and sort  איך לעשות מיון וסינון של נתונים בקומפו לפי מה שהיוזר מזין (לדוגמה סינון על שם ה customer כדי להציג רק לקוחות עם השם הזה)  אם לדוגמה הקומפו מחזיקה פרופרטי (בשם customers) של מערך של Customers והטמפלייט מציגה אותם עם for loop:  ניצור מערך נוסף (למשל filteredCustomers ) שיחזיק רק את ה customers המפולטרים.  הטמפלייט תציג את הפריטים מהמערך המפולטר במקום מהמערך המקורי.  לפי אינפוט שמגיע מהיוזר נשנה כל פעם את רשימת הפריטים במערך המפולטר:  ניצור משתנה פרטי \_filter שיש לו getter, setter (פונקציות קטנות שכל מטרתן לשמור ערך במשתנה / להחזיר את הערך של המשתנה).  ב setter - נבצע קוד שמחליף את הפריטים ברשימת ה filteredCustomers. הוא לוקח את הפריטים מרשימת המקור customers ומפלטר לפי הדרישה, ואת הפריטים המפולטרים שומר ב filteredCustomers.  דוגמה:  נגדיר 2 מערכים. אחד לרשימה המקורית (products) ושני לרשימה המפולטרת. (filteredProducts)    נגדיר משתנה עם getter, setter. כחול- ה setter מבצע קריאה לפונקציה performFilter() שמבצעת את הפילטור על הרשימה המפולטרת. אם listFilter הוא ריק- לא עושה פילטור כלל, שומרת את כל הפריטים ברשימה המפולטרת.    פונקציית הפילטר performFilter(). משתמשת בפונקציה filter() על המערך המקורי, ומחזירה רשימה מפולטרת בשביל ה setter שלמעלה.    נוסיף אתחול של המשתנים בקונסטרקטור. הרשימה filteredProducts מקבלת את כל הפריטים של רשימת ה products, והפילטר סטרינג פה בדוגמה מקבל ערך דיפולטי 'cart'.    נשתמש ברשימה המפולטרת filteredProducts בטמפלייט |
| קינון קומפו - nested component  אם קומפו היא חלק מתצוגה כוללת גדולה יותר- כדאי לקנן אותה.  כדי לקנן קומפו בתוך קומפו אבא - הקומפו חייבת שיהיה לה selector מוגדר בדקורייטור @Component().  אם לא מקננים אותה (למשל אם מציגים אותה רק דרך ראוטינג ולא כקומפו מקוננת- לא חייבים לתת לה סלקטור)  אם התצוגה מורכבת מחלקים שונים - כדאי לפצל אותה לקומפו נפרדות  אם יש צורך לעשות reuse לחלק מסוים מהתצוגה - כדאי לפצל אותו לקומפו נפרדת  אם קומפו  כאשר מקננים קומפו בת בתוך קופו אבא- משתמשים ב @Input, @Output דקורייטור כדי לתקשר ביניהן.  @input  הקומפו מקבלת מידע מהקונטיינר שלה (האבא) על ידי input properties  1. יש לקשט כל פרופרטי שרוצים לקבל מהקונטיינר - על ידי דקורייטור @Input()  2. בקונטיינר - בדיירקטיב של הקומפו הבת- נעשה property binding על הפרופרטי  דוגמה:  קומפו הבת StarComponent.  כחול- שם הדיירקטיב שמסמל את הקומפו. כך נוסיף את הקומפו לקומפו אבא. אדום- פרופרטי rating מקושט בדקורייטור @Input() כדי לקבל את נתוני הרייטינג מהאבא. צהוב- בכל פעם שהקומפו אבא משנה את הערך של הפרופרטי rating אצל הבת קומפו- ה event שנקרא OnChanges (מה lifecycle hooks) נורה, ומפעיל את הפונקציה ngOnChanges שמחשבת מחדש משהו פנימי בקומפו (OnChanges זה lifecycle hook שקורה כאשר ערכים של input properties משתנים).    בקומפו אבא - נוסיף את הקומפו בת כדיירקטיב עם הסלקטור שהגדרנו בקומפו בת (כחול למעלה). אדום- property binding לפרופרטי בשם rating שהגדרנו בקומופ בת (אדום למעלה) |
| קינון קומפו - nested component - המשך  @Output  כדי לשלוח מידע מהקומפו בת לקומפו אבא - הבת חייבת להפעיל אירוע event.  1. יש לקשט כל פרופרטי שרוצים לשלוח לקונטיינר - בדקורייטור @Output()  הפרופרטי חייב להיות event. באנגולר מגדירים event על ידי שימוש בסוג נתונים EventEmmitter<T>. זה סוג שיודע לירות אירועים עם מידע מסוג T. המידע שנשלח נקרא ה event payload.  2.  3. באבא - נעשה event binding על ה event שהגדרנו בקומפו בת.  ניגשים אל המידע שה event שולח (ה payload) דרך האובייקט $event. זהו אובייקט מובנה של אנגולר והוא מייצג את המידע שמועבר בא event. אם זה event של ה DOM – ה $event מייצג אובייקט סטנדרטי של DOM event. אבל אם זה event שכתבתי custom – יכיל את המידע ששלחתי ב eventEmmitter.emit()  דוגמה:  בקומפו בת:  אדום- נגדיר פרופרטי בשם ratingClicked מסוג event (כלומר מסוג EventEmitter<T>). כחול- המתודה onClick מפעילה את האירוע על ידי קריאה לפונקציה emit() של ה EventEmitter. (בהמשך נגדיר event binding על קליק של כפתור כדי להפעיל את המתודה הזאת)    נגדיר על הכפתור בקומפו בת- קריאה למתודה OnClick() בכל פעם שהיוזר מקליק על הכפתור.    בקומפו אבא:  אדום- נגדיר event binding על ה event שהבת שולחת. ורוד- ניגש למידע שה event שולח על ידי מילת המפתח $event. |
| קינון קומפו - nested component - המשך  ngContent ng-content ביחד עם ng-container ngContainer  דיירקטיבים שעובדים ביחד. ngContent משמש "להזריק" תוכן אל קומפו בת. בזמן ריצה – אנגולר תחליף את האלמנט ng-content בתוכן שסיפקנו.  אפשר לספק בקומפו אבא כל תוכן (למשל div ובתוכו עוד HTML), או שאפשר לכתוב את התוכן בתוך ng-container ואז אנגולר תיקח רק את מה שיש בתוך הקונטיינר.  מתי כדאי להשתמש? אם כותבים קומפו' שרוצים להשתמש בה במקומות רבים, ורוצים שיהיה אפשר להחליף לה את התוכן באופן דינמי (למשל רוצים ליצור פאנל ורק להחליף לו את התוכן כל פעם  דוגמה: \*אם יש כמה אלמנטים של ng-content בטמפלייט אחת – מזהים אותם בעזרת אטריביוט select. בדוגמה- יש 2 אלמנטים ומזהים אותם לפי ה css class בקומפו אבא:  קומפו בת: אדום- הסלקטורים של ה ng-content (יש כמה ng-content באותה קומפו אז צריך לזהות כל אחד מהם בעזרת אטריביוט select. בדוגמה- הנקודה לפני השם מסמנת שהתוכן בקומפו אבא יזוהה לפי ה css class שלו.  \*הערה: אם יש רק ng-content אחד לא צריך לזהות אותו עם select אטריביוט    קומפו אבא:  דוגמה בלי ng-container: ירוק- הוספנו קומפו בת עם סלקטור bootstrap-panel (זה הסלקטור שנתנו לקומפו בת) אדום- ה css class בשם heading או body מזהות את האלמנטים מסוג div שבזמן ריצה יחליפו את האלמנט ng-content בקומפו הבת. ה div עם ה css class בשם heading יחליף את אלמנט ng-content עם select=".heading".    דוגמה עם ng-container: ירוק- הוספנו קומפו בת עם סלקטור bootstrap-panel. אדום- נוסיף ng-container אלמנט, ואז אנגולר תיקח את התוכן שלו ותחליף אותו ב ng-content שבקומפו בת: |
| ?  safe navigation operator - question mark - safe traversal operator  אופרטור שמגן מפני NULL או undefined. אם האובייקט שהוא מופעל עליו הוא NULL - אנגולר תחזיר NULL ולא תיגש לפרופרטי (או מתודה) של האובייקט  דוגמה: אם product הוא NULLL - אנגולר לא תנסה לקבל את ה productName של product. רק אם הוא לא NULL - תיגש לפרופרטי productName    שימושי במקרה שמביאים את המידע מהשרת עם HTTP, כי בהתחלה האובייקט הוא NULL עד שהבקשה האסינכרונית מחזירה מידע.  \*אי אפשר להשתמש באופרטור ? עם ngModel data bind.  \*לא כדאי להשתמש באופרטור ? אם יש הרבה גישות לאותו אובייקט בטמפלייט. לדוגמה- טופס שלם שמציג נתונים של אובייקט product.  במקרה כזה - צריך לשקול פשוט לשים \*ngIf על האלמנט אבא, ולבדוק אם האובייקט product הוא NULL. |

XX

|  |
| --- |
| Data Binding  משתמשים ב data binding כדי לקשר בין הלוגיקה (ביזנס) לבין ה VIEW - זה משמש להציג בקלות מידע למשתמש, וכדי להגיב בקלות על פעולות שהיוזר עושה  ה binding תמיד מוגדר בטמפלייט. לא במח'.  יש 3 כיוונים של data bind, ולכל כיוון יש binds משלו:  קוד > VIEW  Interpolation  property binding  class binding  attribute binding  style binding  VIEW > קוד  event binding  קוד > VIEW < קוד (two way)  2 way binding  באנגולר עושים two way data binding על ידי דיירקטיב ngModel ועליו עושים property binding + event binding    data binding זה תקשורת בין הטמפלייט של הקומפו' לקוד behind שלה.  הקוד behind מספק לטמפלייט ערכים להציג,  הטמפלייט מעבירה לקוד behind ערכים שהיוזר סיפק, או מפעילה אירועים (פונקציות) בתגובה לפעולות שהיוזר ביצע (לחיצת כפתור למשל) |
| interpolation - כיוון: מהמח' אל הטמפלייט.  סינטקס: {{template expression}}.  משתמשים בזה כדי:  להוסיף טקסט ל HTML.  לתת ערכים לפרופרטיס של אלמנטי HTML (או של דיירקטיבים)  ניתן לרשום ב template expression של האינטרפוליישן:  פרופרטי של המח'  חישובים  לבצע פעולות פשוטות כמו שרשור סטרינגים  לקרוא למתודה של המח'  temlpate expression - הקוד שרושמים בתוך האינטרפוליישן נקרא template expression.  אנגולר מריצה את הקוד (מבצעת אותו) . את התוצאה שמקבלת היא ממירה לסטרינג ושמה את הסטרינג הזה כטקסט ב HTML, או אם האינטרפוליישן מקנה ערך לפרופרטי של אלמנט - שמה את הסטרינג הזה כערך של הפרופרטי באלמנט DOM.  הקוד מבוצע בקונטקסט של הקומפו'.  מתאים לטקסט שרוצים אותו read only בטמפלייט.  מתאים גם לעשות השמה לפרופרטיס שמקבלים סטרינגים (כמו בדוגמה באדום). אבל אם פרופרטי מקבל סוג נתונים אחר (למשל בוליאני) זה לא מתאים וחייבים property binding  דוגמה:    \*ניתן לקונן interpolation ( {{ xx }} ) בתוך template expression של property binding. עושים את זה אם צריך לקונן template expression אחד בתוך template expression גדול יותר |
| property binding - כיוון: מהמח' אל הטמפלייט  סינטקס: [prop]="template expression"  מאפשר לשנות ערך של פרופרטי של אחד מהשלושה: אלמנט DOM, קומפו של אנגולר, דיירקטיב של אנגולר.  binding target - זה הפרופרטי. תמיד מוקף ב- [ XX ]  binding source - תמיד מוקף בגרשיים- "XX ". מכיל את ה template expression.  דוגמה: אדום- פרופרטי ביינדינג על פרופרטי src של אלמנט img של HTML. כחול- פרופרטי ביינדינג על פרופרטי color של דיירקטיב קסטום שכתבתי בעצמי בשם highlight (שחור) (אשר צובע את הרקע של אלמנטים). תכלת- פרופרטי ביינדינג על פרופרטי בשם numOfRows של קומפוננטה בשם appTable (ירוק).    דוגמאות לשימוש ב property binding: אדום- דוגמה רגילה. כחול- דוגמה איך עושים propery binding על פרופרטי style של אלמנט. זה יותר מורכב- צריך לציין איזה סטייל על ידי שרשור שם הפרופרטי, נקודה, שם הסטייל, נקודה, יחידות    \*פרופרטיס מסוג סטרינג יכולים לקבל Forlation או property binding - זה לא משנה. אבל אם הפרופרטי הוא מסוג אחר (למשל בוליאני) הוא חייב property binding  דוגמה: 2 המקרים יעבדו: |
| attribute binding - כיוון: מהמח' אל הטמפלייט  סינטקס: [attr.xxxnamexxx]="xxx"  משמש לשנות ערכים של אטריביוטים של אלמנטים, כאשר לאטריביוט הזה אין פרופרטי באלמנט שב DOM (נדמה לי)  בגלל שאין פרופרטי ב DOM- אי אפשר לעשות property binding  דוגמה: אדום (לאלמנט DOM של <td> אין פרופרטי שמייצג את האטריביוט colspan ) |
| class binding - כיוון: מהמח' אל הטמפלייט  סינטקס: [class.xxxxClassName]="xxx"  משתמשים כשרוצים להוסיף/למחוק css class לאלמנט לפי תנאי (true/false)  דוגמה: |
| style binding - כיוון: מהמח' אל הטמפלייט  סינטקס: [style.xxxxStyleName]="xxx"  משמש לשלוט בסטייל של אלמנט ישירות inline ב HTML (ולא דרך css class ולא דרך קובץ style וכו' אלא inline בתוך האלמנט)  אפשר להוסיף ב template expression ביטוי למשל תנאי של true/false  דוגמה: |
| event binding - כיוון: מהטמפלייט אל המח'  סינטקס: (eventName)="xxx"  קומפו מאזינה לפעולות של היוזר דרך event binding.  ניתן להוסיף event binding על הדברים הבאים:  event של אלמנט ב DOM  event של קומפו  event של דיירקטיב  target event - שם ה event של האלמנט. מוקף בסוגריים  template statement -. מוקף בגרש  אם ה event קורה- הקוד שב- template statement יתבצע.    רשימה חלקית של אירועים נפוצים: |
| 2 way binding - כיוון: דו כיווני  סינטקס: [(ngModel)]="xxx"  מתבצע רק על אלמנטים input, select textarea .  משמש לעדכן את התצוגה עם ערכים מהפרופרטיס של המחלקה, וגם מעדכן את הפרופרטי של המחלקה כשהיוזר מבצע שינוי בדף. זה נותן זרימה דו כיוונית של תקשורת - מהמח' אל הטמפלייט ובחזרה.  משתמשים בדיירקטיב בשם ngModel ששייך למודול FormsModule שנמצא ב package שנקראת @angular/ forms ויש לייבא אותו למודול הראשי.    \*חובה להוסיף אטריביוט name כשמשתמשים ב ngModel  \*משמש רק ב template driven forms. אם משתמשים ב reactive forms לא עושים 2 way binding |

XX

|  |
| --- |
| דיירקטיבים directives  דיירקטיב זה הוראות לאנגולר לעשות משהו. הם מאפשרים לשנות את ה DOM - להוסיף/להוריד אלמנטים, או לשנות פרופרטיס של אלמנטים ב DOM כמו למשל לשנות את הסטייל, את ה SRC של תמונה, וכו'. הדיירקטיבים מרחיבים את היכולות המובנות של HTML ו- JS  דיירקטיב הוא תגית HTML מיוחדת. היא לא שייכת ל HTML , אלא תגית שייצרנו בעצמנו (או שאנגולר כבר יצאו עבורנו)  או אטריביוט מיוחד של אלמנט HTML.  אנגולר מגיעה עם סט של דיירקטיבים built in.  יש 2 סוגים של דיירקטיבים:  structural - דיירקטיבים שמוסיפים / מורידים אלמנטים מה DOM. למשל \*ngIf, \*ngFor  atribute - דיירקטיבים שמשנים את האטריביוטים של אלמנטים ב DOM. למשל  אפשר לכתוב custom דיירקטיבים בעצמי ולהוסיף אותם לאפליקציה. (למשל- כל קומפו' היא דיירקטיב)  כשהקומפיילר של אנגולר רואה דיירקטיב בתוך טמפלייט – הוא מחפש את ההגדרה של הדיירקטיב במודול.  לכן- צריך להכריז על כל דיירקטיב במודול.  יש דיירקטיבים מסוג structural directives. הם משנים את המבנה של ה HTML ע"י הוספה / הסרה / שינוי של אלמנטים ב HTML.  structural directives -  לדוגמה: \*ngIf, \*ngFor. הם שייכים למודול BrowserModule של אנגולר. יש לייבא אותו ל- AppModule אם רוצים להשתמש בהם.  \*ngIf- מסיר או מוסיף חלקים לעץ ה- DOM לפי תוצאה של expression. |
| structural directives  השם מתחיל ב \*. זה מסמן דיירקטיב כמבני.  ניתן להוסיף רק דיירקטיב מבני אחד לאלמנט.  \*ngIf  שייך למודול BrowserModule של אנגולר. יש לייבא את המודול ל AppModule אם רוצים להשתמש בהם.  דוגמה עם ngIf then else - (#hagar1, #hagar2 הם משתנים לוקאלים של הטמפלייט שנקראים משתני-רפרנס (template reference variables)    הבדל בין \*ngIf לבין אטריביוט hidden של אלמנט HTML:  hidden הוא אלמנט של HTML. סינטקס לדוגמה: <p hidden>  מכיוון ש hidden הוא פרופרטי של ה DOM, ניתן לעשות עליו property binding ולשמור בו ערך true/false.  ההבדל: אלמנט hidden נמצא ב DOM, אבל לא מרונדר כך שהיוזר לא יראה אותו.  אלמנט עם \*ngIf=false - מוסר מה DOM.  אם עובדים עם כמות קטנה של אלמנטים - לא משנה אם משתמשים ב hidden או ב \*ngIf. אם כמות האלמנטים גדולה- צריך לעשות בדיקה ולראות איך זה משפיע על האפליקציה. כי אולי להוסיף הרבה אלמנטים ל DOM יגדיל אותו ויכביד על הזיכרון, על הביצועים של אנגולר (למשל על המנגנון שעוקב אחר שינויים באלמנטים שב DOM). אבל אולי יאיט יותר את האפליקציה לבנות מחדש את כל האלמנטים האלה בכל פעם, ועדיף לבנות אותם פעם אחת ולשמור אותם ב DOM כ hidden. בקיצור- מומלץ לבדוק ולראות מה עדיף בכל מקרה ומקרה.  ngSwitch + \*ngSwitchCase + \*ngSwitchDefault  דיירקטיב \*ngSwitchCase עובד ביחד עם פרופרטי ngSwitch. זהו פרופרטי חדש שאנגולר מוסיפה לכל האלמנטים ב DOM, ואפשר לעשות עליו property binding.  דוגמה: ורוד- נוסיף ל div האבא- property binding על הפרופרטי ngSwitch של אנגולר ונשמור בו את הערך של הפרופרטי viewMode של המחלקה (אדום). ירוק- נוסיף ל div המקוננים - את הדיירקטיב \*ngSwitchCase עם הערך map/list . תכלת- נוסיף ל div בן - את הדיירקטיב \*ngSwitchDefault. |
| structural directives - המשך  \*ngFor  שם הדיירקטיב: ngForOf (כך הוא נקרא באנגולר, למרות שהסלקטור שלו הוא \*ngFor)  דוגמה: אדום- מילת המפתח let יוצרת משתנה של הטמפלייט שנקרא product. ניתן לגשת למשתנה הזה בסקופ של האלמנט או הבנים שלו. תכלת- מילת המפתח of-    הדיירקטיב \*ngFor חושף כמה דברים שימושיים, כמו אינדקס, איבר ראשון ואחרון, שורות זוגיות ואי זוגיות וכו'. איך יודעים את כל הערכים שהוא חושף? נכנסים לאתר אנגולר ומחפשים את הדיירקטיב ngForOf, תחת local variables נמצא אותם.  סינטקס: <p \*ngFor="let x of xxx; exportedVariableName as localVariableName > כאשר exportedVariableName הוא שם המשתנה שהדיירקטיב חושף, ו localVariableName זה ה alias שנתתי לו בטמפלייט שלי.  דוגמה: שימוש באינדקס (משתנה מסוג מספר)    דוגמה: שימוש ב even (משתנה מסוג בוליאני)    trackBy  משמש לשפר ביצועים במקרה של רשימות ארוכות או רשימות עם אובייקטים גדולים  כשמשתמשים ב ngFor - אנגולר מייצרת אלמנט DOM חדש עבור כל איבר במערך.  לפעמים צריך לטעון מחדש מערך מהשרת. גם אם המידע החדש זהה לישן - אנגולר תזהה (ובצדק) שהאיברים במערך התחלפו , והיא תייצר אלמנטי DOM חדשים עבור כל האיברים החדשים במערך, ותמחק את כל אלמנטי ה DOM שמייצגים את האיברים הישנים.  אם הרשימה ארוכה - זה יכול להאיט את האפליקציה. (גם במקרה של אובייקטים מאוד גדולים)  אפשר להגיד לאנגולר: אל תתייחסי לאובייקטים האלה כחדשים, אלא כישנים, ותשמרי על האלמנטי DOM שלהם, אל תיצרי להם חדשים. זה יכול לשפר ביצועים.  איך עושים את זה?  אומרים לאנגולר לעקוב אחרי אובייקטים בעזרת פונקציית מעקב.  בדיפולט אנגולר עוקבת אחרי אובייקטים לפי המקום שלהם בזיכרון (זה נקרא identity)  פונקציית המעקב תדרוס את ההתנהגות הזו, ותגיד לאנגולר מתי אובייקט X זהה לאובייקט Y  דוגמה: תכלת- נוסיף trackBy ל- ngFor ונספק לו את הפונקציה trackCourse (הדוגמה היא מערך של קורסים). שים לב - יש לספק שם של פונקציה ולא להפעיל אותה (לא רושמים סוגריים)    נגדיר את המתודה trackCourse - הפונקציה מקבלת 2 פרמטרים: אינדקס, ואת האובייקט. ומחזירה - את המזהה של האובייקט. עכשיו- במקום לעקוב אחרי אובייקטים (לזהות אותם) לפי המיקום בזיכרון (שזה אומר ש 2 אובייקטים הם שונים אם הם שמורים במקום שונה בזיכרון) - היא תעקוב אחרי אובייקטים לפי הפרופרטי id של האובייקט (שזה אומר שהיא תחשיב 2 אובייקטים כזהים אם יש להם אותו id)    דוגמה: ניקח 2 אובייקטים שונים בזיכרון, אשר זהים מבחינת התוכן:  אובייקט ישן, כתובת בזיכרון: מזא"ה 16, המידע שהאובייקט מכיל: id=1, name='aaa'.  אובייקט חדש: כתובת בזיכרון: מזא"ה 40, המידע שהאובייקט מכיל: id=1, name='aaa'  בעזרת ה trackBy נכריח את אנגולר להתייחס אליהם כאל אובייקט יחיד, ואז היא תמשיך להשתמש ב dom element של האובייקט הישן - עבור החדש.  יש! חסכנו שינויים ב DOM!  הערה: אפשר לראות ב F12 בדפדפן שכאשר אנגולר מייצרת אלמנטים חדשים ב DOM הם מתרפרשים בסגול, לעומת השימוש ב trackBy כאשר האלמנטים לא מתרפרשים ואין צבע סגול מהבהב |
| structural directives - המשך  change detection - מנגנון המעקב (track) אחרי שינויים של אנגולר \*צריך לערוך את הנושא הזה  לאנגולר יש מנגנון של change detection.  אם קורים הדברים הבאים, אנגולר בודקת את המידע של הקומפו (המחלקה) ואם הוא השתנה - מעדכנת את ה DOM ודרכו את מה שהיוזר רואה.:  - אם היוזר לוחץ על כפתור, מזיז עכבר, או מייצר איזשהו dom event  - אם מסתיימת בקשת AJAX  - אם או פונקציית טיימר מסתיימת  נדמה לי שזה קורה כך: נורה אירוע. אם הקומפו רשומה לאירוע - ה event handler מופעל. הוא עלול לשנות את המידע של הקומפו (ה state).  אחר כך: אנגולר בודקת ואם יש שינויים ב state של הקומפו - היא מעדכנת את ה DOM  הדפדפן בתורו מעדכן את התצוגה למשתמש מתוך ה DOM. |
| structural directives - המשך  למה מוסיפים \* לפני שם הדיירקטיב?  כאשר מוסיפים \* לפני הדיירקטיב - זה מה שאנגולר עושה מאחורי הקלעים:  עוטפת את האלמנט באלמנט ng-template ועושה עליו property binding לדיירקטיב ngIf לדוגמה.  דוגמה:  לפני (עם \*)    אחרי: אנגולר עטפה את האלמנט עם ng-template ועשתה עליו בכחול property binding לדיירקטיב ngIf עם התנאי. אחר כך גם הוסיפה באדום property binding לחלק ה else עם שלילה של התנאי הכחול. |
| attribute directives  ngClass  מוסיף css class לאטריביוט class של אלמנט.  הערה: שם ה css class חייב להופיע בתוך גרשיים  הערה: ניתן במקום זה לעשות class binding - ([class.xxxclassName=true/false). אם יש כמה classes כדאי להשתמש ב ngClass    ngStyle מוסיף css style לאטריביוט style של אלמנט.  הערה: נדמה לי ששם ה סטייל חייב לבוא בתוך גרשיים    הערה: אפשר במקום זה להתשמש ב style binding - מה שנוח לנו. דוגמה: |
| קסטום דיירקטיב custom directive  משמש לשנות את המראה או התנהגות של אלמנט HTML של ה DOM  איך ליצור עם cli: ng g directive XXX  מחלקה שמעוטרת עם דקורייטור @Directive  דוגמה:  1. ניצור מח' לדיירקטיב ונעטר אותה עם דקורייטור @Directive. ניתן לו סלקטור. אם רוצים להשתמש כאטריביוט של אלמנט HTML - יש לרשום את שם הסלקטור בתוך סוגריים מרובעים (ירוק)    2 נכריז על הדיירקטיב במודול:    3. נוסיף את האטריביוט לאלמנט HTML:    חוקים:  הסלקטור יירשם בד"כ בקאמלקייס (הסלקטור זה שם האטריביוט ב HTML ). אם יש סוגריים מרובעים מסביב לשם- ישמש כאטריביוט. אם לא- ישמש כאלמנט HTML בפני עצמו.  כדי לקבל גישה לאלמנט שבו הדיירקטיב חי, נשתמש באינג'קשן בקונסטרקטור של ElementRef. אנגולר תדע להזריק שם את האלמנט שאליו הוספנו את הדיירקטיב.  ניתן לגשת אל ה ElementRef בקונסטרקטור, אבל יותר עדיף ב ngOnInit.  לא מומלץ לעשות מניפולציות ישירות על הDOM מה elementRef !!! יש מקרים (מתקדמים) שזה לא יעבוד. עדיף להשתמש דרך כלי העזר renderer.  חשוב! לדיירקטיב אין את כל ה hooks של קומפו'. אבל יש לו את ngOnInit() |
| קסטום דיירקטיב - המשך  ElementRef  כדי לקבל את האלמנט שבו הדיירקטיב חי (הפניה לאובייקט ה HTML) - נעשה DI ל ElementRef:  ElementRef - זהו אובייקט שיש לו פרופרטי יחיד nativeElement שמחזיק את אלמנט ה HTML ב DOM.  @HostListener  כדי לכתוב מתודה שתהיה event handler לאירוע שקורה באלמנט - צריך לכתוב מתודה בדיירקטיב ולקשט אותה עם הדקורייטור @HostListener. הדקורייטור הזה מאפשר לנו להירשם לאירועים שקורים בDOM ELEMENT. כל אירוע שהאלמנט יורה- ניתן להירשם אליו.  הדקורייטור מקבל פרמטר סטרינג- שם ה event (למשל blur, click וכו'), ופרמטר אופציונלי- מערך של ארגומנטים שהאירוע מספק.  דוגמה:    @Input  ניתן להעביר מידע לדיירקטיב. כותבים פרופרטי במח' הדיירקטיב ומקשטים אותו עם @Input דקורייטור. ניתן לכתוב הרבה @Input פרופרטיס. לדוגמה: דיירקטיב בשם appFormat עם אינפוט בשם format נרשום כך:    הערה: אם יש רק פרופרטי אחד - כדאי לתת לו alias עם שם הדיירקטיב (לדוגמה @input('appFormat')) ואז אפשר לעשות פרופרטי ביינדניג על הדיירקטיב כך:    דוגמה לדיירקטיב שמאפשר להכניס רק מספרים לתיבת טקסט: |

XX

|  |
| --- |
| template reference variables - משתנים של הטמפלייט  סינטקס: הכרזה: #xxxName , שימוש: xxxName  כדי שאלמנט או דיירקטיב יוכל לגשת לאלמנט אחר שקיים בטמפלייט - יוצרים משתני reference שמספקים reference לאלמנט אחר בטמפלייט.  דוגמה:    כדי לגשת למשל לערך שנבחר ב select נעשה: myValue.value  עוד דוגמה:    נדמה לי שכדי לגשת לערך של האינפוט נרשום: myVariable.value |

|  |
| --- |
| XXX <form> forms  איך אנגולר עובדת עם טפסים forms  \*יש לי קובץ Angular 6 forms עם פירוט נוסף   1. נעשה אימפורט למודול FormsModule 3. נוסיף תגית <form> לקומפו.   \*לא צריך להוסיף דיירקטיב ngForm ל <form>. אנגולר מוסיפה אותו אוטו' מאחורי הקלעים  זה מייצר אובייקט מסוג NgForm ל <form>. ל NgForm יש למשל פרופרטי errors שמפרט את כל השגיאות של הטופס, או פרופרטי valid בוליאני. יש לו גם פרופרטי Controls (אוסף של אובייקטים מסוג FormControl שמייצגים כל קונטרול בטופס)   1. חשוב! נוסיף novalidate לתגית <form> כדי שהדפדפן לא יבצע ואלידציות משלו. זה ישאיר לאנגולר את הכובע של הואלידציות 2. נשמור את הדיירקטיב במ במשתנה טמפלייט לוקאלי אם רוצים לגשת למשל לפרופרטי valid שלו 3. נוסיף תגית ngModel לכל קונטרול בטופס - input, select, textarea - אם רוצים שאנגולר תעקוב אחריו ותעשה לו binding    * זה מוסיף אובייקט מסוג FormControl עבור האלמנט. האובייקט יתווסף לאוסף Controls של האובייקט NgForm 4. \*חייב להוסיף name אטריביוט 5. \*אפשר (לא חייבים) לעשות עליו binding 6. \*ניתן לשמור את הדיירקטיב במשתנה טמפלייט לוקאלי כדי לגשת לפרופרטי valid שלו, או ל touched, dirty שלו (כדי להציג שגיאות ואלידציה). ניתן לגשת לפרופרטי errors שלו (אובייקט שכל שגיאת ואלידציה מיוצגת כפרופרטי) 7. 4. נוסיף לכל קונטרול בטופס אטריביוטים של ואלידציה. למשל- required, min-length, pattern |
| <form> forms - המשך  דוגמה:  אדום- נשמור את הדיירקטיב ngForm במשתנה טמפלייט לוקאלי בשם frm1.  כחול- נעביר את המשתנה כארגומנט לפונקציה submitForm(). היא תוכל לבדוק את הפרופרטי valid של המשתנה. אם לא כל הטופס ואלידי - הפרופרטי יחזיר false והפונקציה תדע להציג שגיאות למשתמש  ירוק- נוסיף דיירקטיב ngModel לאלמנט <input> - זה יגרום לאנגולר לייצר אובייקט מסוג FormControl (שיש לו שגיאות, מצבי ואלידציה, ו value) ולהוסיף אותו לאוסף של FormControls ששמור בפרופרטי Controls של אובייקט ה NgForm שאנגולר ייצרה אוטו' עבור האלמנט <form>  ורוד- נשמור את הדיירקטיב ngModel (כלומר- את האובייקט FormControl שאנגולר מייצרת עבור הדיירקטיב הזה) במשתנה מקומי בשם username.  תכלת- נציג בדפדפן את השגיאות ואת מצב הקונטרול:  ניגש למשתנה username - אל הפרופרטיס שלו valid, touched, dirty שמציגים את מצב הואלידציה (touched - אם היוזר עשה פוקוס לקונטרול הזה. dirty - אם היוזר שינה את הערך שלו(. ניגש גם אל הפרופרטי errors (אובייקט שמכיל פרופרטי עבור כל שגיאת ואלידציה. אם אין שגיאות - הוא Null)  צהוב - אטריביוטים של ואלידציה ב HTML5    המשך דוגמה- מה שיופיע בדפדפן: תכלת - נדפיס את המצבים של ה FormControl - valid, touched, dirty. וגם נדפיס את השגיאות. |
|  |
|  |

XX

|  |
| --- |
| Pipes  משמשים לשינוי של ערכים לפני שהם מוצגים ליוזר (נקרא transformation)  למשל- שינוי פורמט של תאריך. הצגת מספר עם 2 ספרות אחרי הנקודה. וכו'  יש כמה pipes שמגיעים built in עם אנגולר:  date - לעבודה עם תאריכים  number, decimal, percent, currency - לעבודה עם מספרים (currency - מוסיף את המטבע המקומי (נדמה לי כפי שמוגדר בדפדפן) ב-3 אותיות. כמו USD או EUR)  upperCase, lowerCase - לעבודה עם טקסט  json, slice - לעבודה עם אובייקטים (json מציג את תוכן האובייקט כסטרינג בפורמט json. עוזר לדבג) (slice - בוחר סט של אלמנטים ממערך (נדמה לי))  אפשר לבנות pipe קסטום משלי.  אפשר להשתמש ב pipes באינטרפוליישן וגם ב propert binding    ניתן לשרשר פייפים:    יש פייפים שמקבלים פרמטרים. מציינים פרמטרים עם נקודותיים אחרי שם הפייפ, ואז את הערך של הפרמטר.  אפשר לשרשר כמה פרמטרים:  בדוגמה- פייפ ה currency מקבל 3 פרמטרים: קוד המטבע (דולר), סטרינג שמציין שיוצג הסמל $ ולא האותיות USD, ואיך להציג את המספר (לפחות ספרה אחת לפני הנקודה ובדיוק 2 ספרות אחרי הנקודה    דוגמה לתוצאת הפייפ currency שבדוגמה מעל: (כל מספר כולל 2 ספרות אחרי הנקודה בדיוק, וסמל $(    \*אנגולר לא ממליצה להשתמש בפייפים כדי לסנן תוצאות ברשימה. זה נותן ביצועים גרועים. במקום זה מומלץ להעביר את הלוגיקה של מיון וסינון אל תוך הקומפו עצמה. לא לעשות זאת עם פייפ. |
| איך לבנות custom pipe  לדוגמה- pipe שמחליף מקף ברווח.  איך ליצור Pipe:  1. כותבים מח' חדשה (ועושים לה export)  2. מקשטים אותה בדקורייטור @Pipe().  נגדיר את שם הפייפ שישמש בטמפלייטים כשנרצה ליישם את הפייפ (בפרופרטי name של הדקורייטור)  3. מממשים את האינטרפייס PipeTransform (נמצא ב @angular/core)  יש לו מתודה יחידה transform() שמקבלת סטרינג פרמטר בשם value - הערך שיש לשנות (לדוגמה המספר "32" שנרצה להציג כ "32.00 $" תוך שימוש בפייפ).  המתודה מחזירה סטרינג - זה יהיה תוצאת הפייפ.  אם רוצים פרמטרים נוספים לפייפ- יש להוסיף אותם כפרמטרים נוספים של המתודה transform()  4. נוסיף את הפייפ לרשימת ה declerations של המודול  דוגמה:  נכתוב Pipe שהופך לדוגמה מקף לרווח. (הפייפ יהיה גנרי ולא יהפוך רק מקף, אלא כל תו שנבחר להעביר לו כפרמטר)  תכלת- מימוש אינטרפייס PipeTransform,אדום- דקורייטור @Pipe(), כחול- שם הדיירקטיב בטמפלייטים, ורוד- מימוש המתודה transform, ירוק- פרמטר נוסף של הפוקנציה מסוג סטרינג (ייצג את התו שיש להפוך לרווח)    שימוש בפייפ בתוך טמפלייט על ערכים שמגיעים מתוך הפרופרטי product.productCode: אדום- שם הפייפ, כחול- הערך שיישלח לפרמטר character (בירוק) מהדוגמה למעלה (במקרה הזה- מקף)    נוסיף את הפייפ ל declerations של המודול:    תוצאת הפייפ:  התצוגה בלי הפייפ:    התצוגה עם הפייפ: |

XX

|  |
| --- |
| Services  סרביס משמש ל 2 דברים שונים:  להחזיק מידע / פעולות שרוצים לחלוק בין קומפו' רבות. כמו למשל UserService שיחזיק מידע על היוזר שעשה לוגין.  לבצע פעולות מסובכות או לוגיקה שרוצים להוציא מהקומפו. כמו למשל ProductsDataService שבצע פעולה של הבאת מידע של products מהשרת.  איך ליצור סרביס:  1. נכתוב מחלקה חדשה (בקונבנציה נקרא לה XxxService. ולקובץ xxx.service.ts)  2. נגדיר אותה כסרביס על ידי הוספת דקורייטור @Injectable()  3. נחליט אם הסרביס צריך להיות משותף לכל האפליקציה (קומפו + סרביסים אחרים), או רק לקומפו' מסוימת (ולבנות שלה).  אם הוא משותף - נרשום אותו ב root injector על ידי פרופרטי providedIn של הדקורייטור עם הערך "root"  אם הוא ברמת קומפו מסוימת - נרשום אותו ב component injector על ידי פרופרטי providers של הדקורייטור של הקומפו @Component  4. נגדיר אותו כ dependency של מח' שצריכות להשתמש בו. מגדירים dependency ב typeScript על ידי הוספת פרמטרים לקונסטרקטור של מח'.  דוגמה:  נכתוב את הסרביס (המח'). אדום- נקשט אותה עם דקורייטור @Injectable . כחול- רושמים את הסרביס ב root injector של האפליקציה על ידי פרופרטי providedIn עם הערך "root"  \*\*\*הסינטקס הזה חוקי רק מאנגולר 6 ומעלה. לפני כן - היה צריך לרשום סרביס ב metadata של המודול.    נוסיף פרמטר מסוג ProductService לקונסטרקטור של הקומפו ProductListCompo (זה מוסיף לקומפו dependency injection) וה injector של אנגולר ידע לספק את הסרביס לקומפו)    דוגמה איך לרשום סרביס ל component injector  נרשום את הסרביס ברשימת ה providers של הקומפו עצמה בדקורייטור @Component. דוגמה: רושמים את סרביס ProductService כסרביס של ה component injector של הקומפו ProductListCompo    \*\*\*הערה חשובה \*\*\*  לפני אנגולר 6 - אי אפשר היה לרשום ב root injector בדקורייטור של הסרביס. במקום זה רשמו בפרופרטי providers של הדקורייטור @NgModule() של המודול עצמו  \*עדיף להשתמש בסינטקס החדש אם אפשר, כי הוא יותר טוב ל tree shaking (תהליך שבו אנגולר זורקת קוד שלא בשימוש כדי להקטין למינימום את ה bundles (הקבצים) של הדיפלויימנט) |
| Services - המשך - מידע כללי  למה צריך סרביסים?  מה עושים עם מידע או לוגיקה שלא קשורים ל view ספציפי?  מה עם מידע שרוצים לחלוק בין כמה קומפו?  בונים סרביס.  לדוגמה- מידע על היוזר שעשה לוגין. נרצה לחלוק את המידע הזה בין הרבה קומפו.  לדוגמה- סרביס שאחראי להביא מידע מהשרת. נרצה להוציא את הלוגיקה הזו מהקופו לסרביס נפרד, שנותן שירות לקומפו.  לדוגמה- סרביס שאחראי לכתוב לוגים. זו לוגיקה שנרצה שהרבה סרביסים אחרים ישתמשו בה.  סרביס זו מחלקה עם מטרה מסוימת.  למשל- מטרה כללית שלא מתאימה רק לקומפו מסוימת (כמו כתיבת לוגים).  למשל- מטרה של החזקת מידע שמשותף לקומפו' רבות (כמו מידע על היוזר הנוכחי שעשה לוגין לאפליקציה)  למשל- מטרה של הוצאת לוגיקה מסובכת מהקומפו (כמו הבאת מידע מהשרת לקומפו)  זה עוזר לשמור את הקומפו' קטנות מבחינת נפח קוד. יותר קל לבדוק את הקוד ולשלוט בו, ולעשות לו reuse.  כדי להוסיף את סרביס הזה (Inject) לקומפו שזקוקות לו - משתמשים במנגנון שנקרא dependency injection.  יש סרביסים שרק מספקים פעולות כמו סרביס שמבצע הבאת מידע מהשרת. לסרביסים כאלה בדרך כלל אין פרופרטיס פבליק שמספקים מידע, רק מתודות שמבצעות פעולות.  סרביסים אחרים חולקים מידע בין קומפו, ויהיו להם פרופרטיס. ואולי לא יהיו להם מתודות בכלל. |
| Services - המשך - מידע כללי  הסבר כללי איך סרביסים עובדים באנגולר  services באנגולר עובדים עם עם dependency injection.  dependency injection זו שיטה שבה מחלקות לא מייצרות אובייקטים חדשים שהן צריכות, אלא מקבלות אותן ממקור חיצוני. למה זה טוב? זה חוסך משאבים (זיכרון, זמן), לדוגמה- בהרבה מקרים לא צריך שכל מח' תייצר אובייקט חדש לעצמה, הן יכולות לחלוק את אותו אובייקט. זה גם טוב לבדיקות.  האובייקטים שהן צריכות נקראים dependency שלהן.  איך אנגולר עושה dependency injection לסרביסים? על ידי injector  ה injector של אנגולר שומר אצלו אוסף של סרביסים והוא מספק אותם (provide או inject) לקומפו שתלויות בהם לפי דרישה  ה injector מחזיק אינסטנס יחיד עבור כל סרביס, הוא לא מייצר אובייקט חדש כל פעם, אלא משתמש באותו אחד בשביל כל הקומפו שמבקשות אותו.  הוא מספק את הסרביס לקומפו בזמן האתחול של הקומפו.  זה מונע איתחול של הרבה אובייקטים מיותרים מאותו סרביס. זה יעיל וחוסך זיכרון ומשפר מהירות ביצועים.  הקומפו מגדירות שהן צריכות סרביס מסוים כך: הן מוסיפות פרמטר לקונסטרקטור שלהן, מסוג הסרביס. כך אנגולר יודעת לספק אותו מה injector.  אנגולר מחזיקה עץ של אינג'קטורים:  ה-root injector משמש לכל האפליקציה. סרביסים שרשומים אצלו זמינים לכל הקומפו' וגם לסרביסים אחרים.  ה- component injector - לכל קומפו יש injector משל עצמה. סרביסים שרשומים ברמה זו זמינים לקומפו ולכל הקומפו בנים שלה. אם רושמים סרביס ברמת הקומפו אז כל פעם שהקומפו מאותחלת - נוצר עבורה אינסטנס חדש של הסרביס. ברמת הקומפו - הסרביס מפסיק להיות סינגלטון (כלומר יכולים להיות באפליקציה הרבה אינסטנסים מאותו סרביס)  בדרך כלל רושמים סרביס ב root injector, אבל יש מקרים שנרצה לרשום ב compo injector  אנגולר מייצרת את האינסטנסים של הסרביסים בעזרת providers. ה Provider - אחראי לייצר אינסטנס חדש מסרביס, ולהחזיר אותו  \* אפשר תמיד לעשות new לסרביס בתוך מחלקה עצמה, אם לא רוצים לקבל את הסרביס המשותף מה injector.  אם יש לי סרביס (לדומגה מחלקה בשם Service1) ואני רוצה להשתמש בה בקומפו' A - אפשר פשוט ליצור משתנה (למשל משתנה בשם svc) מסוג Service1 ולעשות לו new() - כך נקבל אובייקט חדש מסוג Serivce1 בתוך קומפו A:  svc: Service1 = new Service1();  אבל האובייקט ב svc הוא מקומי בקומפו A, אי אפשר לחלוק אותו עם קומפו נוספות.  גם יהיה קשה לעשות mock לסרביס למטרות של בדיקה (לא כל כך ברור לי מה זה, מקווה שיתברר בהמשך)  \*ניתן לרשום סרביס כמה פעמים - עבור injectors של קומפו' שונות.  אם קומפו זקוקה לסרביס - היא צריכה להגדיר אותה כ dependency - כאובייקט שהיא תלויה בו . כך ה injector יודע שהיא זקוקה לו ומספק אותו. היא עושה את זה באמצעות הוספת פרמטרים לקונטסרקטור שלה  בגלל שהסרביס משותף לקומפו שונות - המידע שהוא מחזיק נגיש לכולם (ה properties), והוא יכול לבצע את הפעולות שלו בשירות כולם (כולם יכולים להפעיל את המתודות שלו) |
| Injector  האינג'קטור של אנגולר יוצר אינסטנס של סרביס, ומזריק אותו inject למחלקה שתלויה בו.  בדרך כלל לא ניצור אינג'קטור בעצמי. אנגולר יוצרת בעצמה אינג'קטורים כשהיא מריצה את האפליקציה. למשל- בתהליך האיתחול של האפליקציה (ה bootstrap) אנגולר יוצרת את ה root injector.  לאינג'קטור יש פרוביידרים?  Provider או service provider  אובייקט שמגדיר איך להשיג את ה  פרוביידר משמש להגדיר |

XX

|  |
| --- |
| HTTP Service  אנגולר מגיעה עם סרביס built in שמאפשר לבצע בקשות HTTP לשרת – שנקרא HttpClientService. הוא נמצא ב HttpClientModule אשר נמצא בפקג' - @angular/common/http  הסרביס חושף את המתודות get, post, delete וכו – המחזירות Observable<T>  לכן- יש להירשם subscribe() ל observable כדי לקבל את ה http response.  הסרביס חושף את המתודות הרגילות ב HTTP:    כדי לעבוד עם ה HttpClientService צריך:   1. לייבא את המודול HttpClientModul אל תוך המודול של האפליקציה שלי 2. ליצור סרביס משלי (לדוגמה CustomersDataService) שתפקידו לבצע גישת HTTP לשרת תוך שימוש ב HttpClient סרביס. (כדאי להחצין את הפנייה לשרת backend לסרביס נפרד ולא לעשות זאת בתוך הקומפו) 3. להוסיף לסרביס שלי dependency ל HttpClient 4. להוסיף לדוגמה מתודה לסרביס שלי getXXX - שמחזירה Observable<T> המתודה הזאת תקרא למתודה get של ה HttpClient סרביס. 5. חשוב - לעשות error handling על ה get() של ה HttpClientService, כי הרבה דברים יכולים להשתבש בפניה לשרת backend. 6. בקומפו' – שצריכה מידע מהסרביס CustomerDataService.get() למשל – נירשם ל observable שקיבלנו חזרה מה httpClientService. 7. יש לספק ל subscribe() מתודה לטפל במידע שה observable מחזיר 8. כדאי לספק גם פונקציית error handling כדי לטפל בשגיאות אם היו 9. אם רוצים לדבג את מה שה observable מחזיר - כדאי להשתמש באופרטור tap   דוגמה למודולים (של JS) ול imports שצריך לעשות כדי לעבוד עם Observable ועם אופרטורים:  import { Observable, range } from 'rxjs';  import { map, filter, catchError, tap, take } from 'rxjs/operators'; |
| דוגמה לשליחת בקשת HTTP לשרת: אדום- מייבאים את מודול HttpClientModule למודול הראשי של האפליקציה   1. נייבא את ה HttpClientModule – נוסיף import למודול הראשי של האפליקציה 2. ניצור סרביס שרוצה לפנות לשרת ב HTTP – ונוסיף את הסרביס HttpClient כ dependency   דוגמה: נייבא את המודול HttpClientModule לאפליקציה    ליצור סרביס חדש MechironDataService שיעטוף את הפניות לשרת backend (לא כדאי לשים את הקוד הזה בקומפו ישירות). אדום- נוסיף dependency על ה HttpClient סרביס.    לייבא לסרביס החדש שיצרתי את האובייקטים Observable, והאופרטורים שאני רוצה להשתמש בהם - מספריית rxjs:    לכתוב בתוך מתודה של הסרביס שלי את הפניה ל HttpClient.get() אשר מחזיר לי observable<T>. כדאי להוסיף אופרטור catchError() שמטפל בשגיאות. מוסיפים אופרטורים בתוך מתודה pipe(). (בדוגמה- לא פניתי למתודה httpClient.get() אלא יצרתי Observable חדש בעזרת מתודה range() של rxjs, סתם בשביל הדוגמה). לגבי catchError() יש הסבר בהמשך בחלק על אופרטורים של rxjs |
| Http get method  המתודה מקבלת פרמטר – url – שמגדיר לאן בקשת ה HTTP תישלח, ומחזירה Observable<T> או נדמה לי גם Observable לא גנרי  יש מתודה get<T> גנרית – שמאפשרת להגדיר ב T מה סוג המידע שחוזר ב JSON פורמט מהשרת. המתודה get תמפה (map) אוטו' את ה response שהיא מקבלת – אל האובייקט שהגדרנו ב T. לכן- מחזירה את T ולא response גולמי  חשוב להוסיף למתודה get - error handling כי הרבה דברים יכולים להשתבש  דוגמה:  קריאה למתודה get של HttpClient. אדום- סוג נתונים שהוגדר ל T הוא void, לכן- בכחול- המתודה מחזירה Observable of void |
| Exception Handling  הרבה דברים יכולים להשתבש כשפונים לשרת backend. למשל- בקשה לא תקינה, בעיות בתקשורת, וכו  לכן צריך להוסיף טיפול בשגיאות.  חייבים להוסיף טיפול בשגיאות – אחרת נקבל unhandled exception באפליקציה  עושים זאת בעזרת אופרטור catchError()  catchError () – מקבל פונקציה שמקבלת את השגיאה כפרמטר. היא צריכה להחזיר Observable חדש (אני לא בטוחה אם חייבת).  \*אין אפשרות להשתמש ב try catch רגיל – בגלל שזה אסינכרוני. התשובה תגיע בעתיד. ועד אז – כבר סיימנו לבצע את הקוד שב try ולא נכנסנו ל catch. אבל קרתה שגיאה בעתיד (ב אובזרבבל). איך מטפלים בה?  בעזרת אופרטור catch()  catchError – תופס כל שגיאה  דוגמה:  בדוגמה יש שימוש בפונקציה throwError("xxx") של rxjs. יש הסבר מפורט עליה בחלק של ה rxjs בהמשך. מה שהיא עושה זה להחזיר observable שלא פולט נתונים כלל, רק שגיאה יחידה. זה טוב לטיפול בשגיאות: האופרטור catchError() תופס את השגיאה המקורית שנזרקה, מטפל בה (רושם לוגים וכו'), ושוב זורק שגיאה כך שהנרשמים יקבלו שגיאה וידעו לפעול בהתאם. |
| Windows Authentication  איך לשלוח credentials בבקשות של HttpClient סרביס:  צריך להוסיף ב options של הבקשה – את האופציה "with credentials":  דוגמה:    אם לא רוצים לחזור על זה בכל בקשה – אפשר להוסיף interceptor – שיתפוס כל בקשה שיוצאת – ויוסיף לה credentials:  דוגמה (אנגולר 6):  נכתוב סרביס שמממש את האינטרפייס HttpInterceptor. יש לו מתודה יחידה intercept()    נרשום את הסרביס הזה במודול הראשי של האפליקציה. (אי אפשר ב providedIn של הדקורייטור @Injectable() של הסרביס, כי צריך לעשות את זה ככה (לא מבינה מה כתוב פה: |
| Windows Authentication - המשך  איך לקנפג את השרת:  1. צריך להוסיף package של CORS לפרוייקט web api  2. לקנפג במתודה Register() של מחלקה WebApiConfig:    3. בכרום בקשות OPTIONS נכשלות (אלו בקשות שכרום שולח לפני POST ואולי גם PUT) כי השרת מחזיר תשובה בלי headers מסויימים שה כרום מצפה להן.  נוסיף בקובץ Global.asax את הטיפול ב event לבקשות נכנסות: |

XX

|  |
| --- |
| מודול של אנגולר angular module angular modules  אפליקציית אנגולר מורכבת ממודולים. מודולים עוזרים לארגן אפליקציה ולחלק אותה לפי נושאים.  בכל אפליקציה יש לפחות מודול אחד. זה המודול הראשי.  לפי הקונבנציה קוראים לו AppModule  נטען לטעון מודולים בעליית האפליקציה, או בצורה lazy אם זה נדרש.  מה זה מודול?  זו מחלקה שמקושטת בדקורייטור @NgModule()  בדקורייטור מספקים מידע מטא-דאטא אודות המודול (הפונקציית דקורייטור מקבלת אובייקט עם פרופרטיס. פירוט בהמשך)  כל קומפו, דיירקטיב או פייפ- חייבים להיות שייכים למודול אחד ויחיד בלבד.  מודול הוא כמו קופסה שצריכה לעמוד ברשות עצמה ולספק את כל צרכיה.  טמפלייט רזולושן- זה מושג שמשמעותו שטמפלייט (של קומפו) ששייכת למודול מסוים- המודול הזה צריך לספק את כל צרכיה.  לכן לדוגמה: אם קומפו מסוימת של מודול 1 משתמשת בדיירקטיב A - אז דיירקטיב A חייב להיות מוכרז במודול 1, או לחילופין מודול 1 יכול לייבא את מודול 2 אשר מייצא את דיירקטיב A. אם שני אלה לא מתקיימים- נחטוף שגיאה.  אם קומפו משתמשת לדוגמה בדיירקטיב \*ngIf (אשר מוגדר במודול BrowserModule) - צריך לייבא את BrowserModule למודול שלנו. אחרת נחטוף שגיאות.  מה מודול עושה:  מגדיר (מכריז) קומפו, דיירקטיבים ופייפים.  מאתחל (עושה bootstrap) לקומפו' הראשית (ה root) של האפליקציה (זה רק במודול הראשי של האפליקציה)  יכול לייצא קומפו, דיירקטיבים ופייפים, וגם מודולים אחרים, כך שמודולים אחרים יכולים לייבא ולהשתמש בהם.  יכול לייבא מודולים אחרים  יכול לרשום סרביסים באינג'קטור (מאז אנגולר 6 רושמים סרביסים דרך דקורייטור @Injectable() של המחלקה ולא דרך הפרופרטי providers של המודול)  מודולים נפוצים של אנגולר:  RoutingModule  BrowserModule - מכילים דיירקטיבים כמו \*ngIf' \*ngFor  FormsModule - מכיל דיירקטיבים כמו NgModel  מודולים צד 3 נפוצים:  MaterialDesign  איך לייצר מודול חדש עם angular cli:  ng g m moduleName  \*הבדל בין מודול של אנגולר למודול של JS  \*אנגולר גם משתמשת במודולים של JS. מודול ב JS זה פשוט קובץ. ניתן לייצא מהמודול מח', פונקציות, משתנים וכו'. ניתן לייבא אותם אחר כך במודולים (קבצים) אחרים. זה קל ומודולרי ואפשר לבנות קוד נוח וקריא בעזרת מודולים (קבצים) מרובים של JS. |
| דוגמה לאפליקציה המחולקת למודולים:  מודול אדמין - מכיל את כל הקומפו' שקשורות לאדמין של האפליקציה, ואת הדיירקטיבים והפייפים שהם צריכים.  עושה להם declare.  לא עושה להם export כי לא צריך  מייבא את CommonModule אם צריך דיירקטיבים כמו ngIf, ngFor (אסור לייבא את BrowserModule, ובכלל - הדיירקטיבים האלה נמצאים ב CommonModule וה BrowserModule מייבא את ה CommonModule וגם מייצא אותו). מייבא את RouterModule של אנגולר אם הוא צריך.  מייבא את מודול shared אם הוא צריך משם משהו  וכו'.  מודול רכישות - מכיל את כל הקומפו' דיירקטיבים ופייפים של רכישות.  עושה declare, export, import בדומה למודול אדמין  מייבא את מודול shared אם הוא צריך משם משהו  מודול shared - מכיל דיירקטיבים ופייפים שמשתמשות בהם קומפו מ- 2 מודולים: קונפיגורציה ורכישות.  עושה להם גם declare וגם export.  מודול ראוטינג -  לא עושה declare לכלום,  מייבא את מודות RouterModule של אנגולר ומקנפג את כל הראוטים של האפליקציה  לא עושה import לקומפו' ממודולים אחרים. הוא לא צריך כי אין לו קומפו שמשתמשות בזה.  בשביל לקנפג את הראוטים לא צריך לעשות imoprt למודולים שהן מוכרזות בו  מודול core - אם זה לפני אנגולר 6 - משמש להוסיף providers לכל הסרביסים של האפליקציה שנרשמים ב root injector וצריך להוסיף אותם רק פעם יחידה (אסור לייצא סרביסים ואסור לעשות להם provide יותר מפעם אחת. זה יגרום לכך שהם לא יהיו סינגלטונים)  מודול ראשי (root module) - נקרא בדרך כלל AppModule.  מייבא את המודולים: ראוטינג, רכישות, אדמין, core. לא מייבא את מודול shared אלא אם הקומפו שלו זקוקות לו. מייבא את המודול BrowserModule (חובה בשביל אפליקציות שרצות בדפדפן) |
| דוגמה להכרזה על מודול הראשי של האפליקציה (בקונבנציה קוראים לו AppModule): ירוק- הרכזה על המחלקה AppModule, ורוד- מקשטים אותה בדיירקטיב @NgModule, אדום- declerations - הפרופרטי של ה דקורייטור @NgModule אשר מכריז על כל הקומפו, דיירקטיבים ופייפים שמוגדרים במודול הזה. כחול- כל מה שהמודול מייבא: BrowserModule (דיירקטיבים \*ngIf, \*ngFor למשל מוגדרים במודול הזה), מודול FormsModule (שבו למשל מוגדר הדיירקטיב NgModule), RouterModule ועוד. תכלת- מגדיר את האתחול של האפליקציה (ה bootstrap שלה) שתתחיל את הקומפו הראשית של האפליקציה- ה AppComponent. |
| פירוט הפרופרטיס של הדקורייטור @NgModule() - אשר מגדיר את המטא-דאטא עבור המודול:  \*כל פרופרטי מקבל מערכים.  bootstrap- מוגדר רק במודול הראשי, וחייב להיות מוגדר במודול הראשי.  מגדיר את הקומפו שתאותחל בעליית האפליקציה. (הדיירקטיב של הקומפו הזאת (הסלקטור שלה) יופיע בקובץ index.html)  כל אפליקציה חייבת להגדיר bootstrap במודול הראשי של האפליקציה, ורק במודול הראשי.  declerations - מכיל את כל הקומפו, דיירקטיבים ופייפים של המודול. כל קומפו, דיירקטיב ופייפ- יכולים וחייבים להיות מוגדרים על ידי מודול אחד בלבד. קומפו שמוכרזים במודול מסוים- אומרים עליהם שהם שייכים למודול הזה.  לא יכול להכיל דברים אחרים כמו מודולים, סרביסים וכו'. רק קומפו, דיירקטיב ופייפ.  אין להכריז על קומפו, דיירקטיב, פייפ- ביותר ממודול אחד.  אם צריכים אותם מחוץ למודול- על המודול לעשות להם export. אחרת- בדיפולט הקומפו של מודול הוא פרייבט לאותו מודול, ונגיש רק לקומפו ששייכים לאותו המודול. כדי להפוך קומפו לפבליק- צריך לעשות לו export.  export - ניתן לייצא קומפו, דיירקטיב ופייפ של מודול- כך הופכים אותם לפבליק והם זמינים למודולים אחרים שמייבאים את המודול שלהם.  ניתן גם לייצא מודולים אחרים  \*\*\* ניתן לייצא מודול גם אם לא עשינו לו import (למשל אם רוצים לעשות SharedModule לכל האפליקציה- הוא יכול לייצא את CommonModule (שמכיל דיירקטיבים כמו \*ngIf) - גם בלי לעשות לו import. כך כל מודול באפליקציה שייבא את ה SharedModule - יקבל את ה CommonModule אוטו')  אין ליצא סרביס. (סרביסים שמופיעים ב providers של מודול- הם רשומים באינג'קטור ה root של האפליקציה, ולכן הם זמינים לאינג'קשן לכל מחלקה באפליקציה. לכן אין צורך לייצא אותם, הם כבר זמינים לרוחב האפליקציה)  Imports - מייבא מודולים אחרים כדי להתמש בקומפו, דיירקטיב ופייפים שהם מייצאים. לדוגמה- אם רוצים להשתמש בדיירקטיב \*ngIf שמוגדר במודול BrowserModule - יש לייבא אותו.  כשמייבאים מודול- זה לא מספק גישה למודולים שהוא מייבא!!! רק למה שהוא מייצא. כלומר- מודול לא "יורש" את ה imoprts של מודול אחר.  לדוגמה- אם מודול A מייבא את FormsModule אז קומפו 1 של מודול A יכולה להשתמש בדיירקטיב NgModel של FormsModule.  אבל אם מודול B מייבא את מודול A, אז קומפו 2 של מודול B לא יכולה להתמש בדיירקטיב NgModel של FromsModule. רק אם מודול B מייבא את FormsModule גם כן בעצמו (או שמודול A מייצא את FormsModule)  לסיכום- מודול הוא כמו קופסה, לא כמו עץ של מודולים. הוא יכול לייבא מודולים אחרים, אבל הם לא חושפים לו אוטו' את כל הקומפו שלהם, הם חושפים רק את מה שהם עושים לו export.  providers - רושמים סרביסים לאינג'קטור הראשי (root) של האפליקציה. אבל מאז אנגולר 6 לא משתמשים בזה יותר. יש להשתמש בדקורייטור @Injectable() על הסרביס, ולהגדיר בו את הפרופרטי- providedIn: 'root' - זה ירשום את הסרביס לאינג'קטור הראשי.  ברגע שרושמים סרביס לאינג'קטור הראשי- הוא זמין לשימוש בכל רחבי האפליקציה (ניתן לעשות לו אינג'קשן לכל מחלקה, גם מחלקות ממודולים אחרים), ולכן זה לא תלוי במודולים. לכן גם אין לרשום סרביסים (אם רושמים אותם ב providers של המודול ולא בדקורייטור @Injectable) למודול שמיובא פעמים רבות (כלומר מיובא לתוך מודולים שונים באפליקציה) כי זה כנראה ירשום את הסרביס הרבה פעמים לאינג'קטור וזה לא מה שאחנו רוצים כנראה) |
| BrowserModule - הערה חשובה  כל אפליקציה שרצה בבראוזר - חייבת לייבא את המודול הזה. אבל מותר לייבא אותו רק פעם אחת ויחידה באפליקציה, לכן חייבים לייבא אותו לתוך המודול הראשי של האפליקציה.  מה עושים אם צריכים להשתמש בדיירקטיבים כמו \*ngIf, \*ngFor במודולים לא ראשיים של האפליקציה?  מייבאים אליהם את CommonModule במקום את BrowserModule  CommonModule  הדיירקטיבים \*ngIf, \*ngFor וכו'- מוגדרים ב CommonModule ולא ב BrowserModule, והוא רק עושה להם export. (למעשה BrowserModule מייבא את CommonModule וגם מייצא אותו) |
| forRoot(), forChild()  נדמה לי - כשמייבאים מודול לתוך המודול הראשי (root) של האפליקציה - משתמשים ב forRoot()  כשמייבאים מודול לתוך מודול לא ראשי- משתמשים ב forChile()  מה ההבדל? אפשר להתשתמש ב forChild() כמה פעמים (בכל מודול לא ראשי של האפליקציה), אבל משתמשים ב forRoot() רק במודול הראשי. כך- אם המודול צריך לרשום סרביסים לאינג'קטור הראשי של האפליקציה- הוא יודע לעשות זאת רק כשקוראים ל forRoot() ולא ל forChild(), וכך הוא לא ירשום את אותו סרביס יותר מפעם אחת.  נדמה לי שמודול שמייבאים עם forChild() - חייבים גם לייבא עם forRoot() פעם נוספת במודול הראשי של האפליקציה (לצורך רישום הסרביסים שלו) (אולי זה רק למודולים מסוימים כמו ה RoutingModule שחייב לרשום את הסרביסים שלו route, activatedRoute לאינג'קטור של האפליקציה)  \* לדוגמה- אם רוצים לייצא את הראוטינג למודול נפרד וגם לחלק אותו לתתי מודולים - צריך פעם אחת לפחות לעשות forRoot() ואז גם לעשות לו export כדי שלא יצטרכו לייבא אותו למשל פעם נוספת במודול הראשי של האפליקציה (מה שיגרום ל forRoot() נוסף וזה לא טוב) |

XX

|  |
| --- |
| ראוטינג routing  אנגולר היא SPA - מבוססת על דף יחיד, אשר התצוגה בו מתחלפת. לא מנווטים בין דפים שלמים, אלא בתוך הדף- בין תצוגות.  איך אנגולר יודעת מה להציג? לפי נתיבים - ראוטים - routes.  מקנפגים route לכל קומפו שרוצה להציג את התצוגה שלה על הדף  מוסיפים תפריט ניווט, כפתורים, וכו' - כדי לאפשר ליוזר לנווט בין התצוגות השונות.  מקשרים route לכל כפתור, קישור וכו' בניווט  כאשר היוזר לדוגמה לוחץ על הכפתור - הראוט המתאים מופעל. כאשר הראוט מופעל - אנגולר מציגה את הקומפו' המתאימה  מה ראוטינג מודול עושה:  רושם את הסרביס router, activated route  מכריז על דיירקטיבים של הראוטר (למשל router-outlet, routerLink)  חושף את הראוטים המקונפגים שהגדרנו באפליקציה. |
|  |
|  |

XX

|  |
| --- |
| APP\_INITIALIZER  איך לבצע דברים לפני שהאפליקציה נטענת (תוך כדי תהליך האיתחול של אנגולר bootstrap) – למשל לטעון קובץ קונפיג מראש לפני טעינת האפליקציה.  באנגולר (4 ומעלה) ניתן לכתוב פונקציה שתרוץ בזמן האתחול, מתישהו בתהליך ה bootstrap.  אם הפונקציה הזו מחזירה promise – אנגולר תעכב את טעינת האפליקציה עד שהפרומיס תסתיים. זה שימושי כי זה מאפשר לי למשל לטעון קובץ קונפיג ולחכות עד שייטען, ורק אז לטעון את האפליקציה.  (פרומיס אובייקט מיוחד של JS שמבצע פעולות אסינכרוניות ומודיע לנרשמים כשהפעולה הסתיימה. כשהפרומיס הסתיימה זה נקרא resolve).  איך עושים את זה:  דרך מנגנון ה DI של אנגולר.   1. נכתוב פונקציה בשם XXX באותו קובץ של ה app.module.ts או בקובץ אחר ונעשה לה export. 2. נוסיף למערך providers של הדקורייטור @NgModule() אובייקט מסוג פרוביידר אשר יקבל את המפתח (נקרא DI Token) APP\_INITIALIZER, ואת הפונקציה XXX שכתבנו   הערה: ניתן לכתוב יותר מפונקציית אתחול אחת. במקרה כזה – יש להוסיף 2 אובייקטים של פרוביידר, כל אחד קוראת לפונקציה אחרת. 2 הפונקציות ירוצו במקביל, ואנגולר תחכה עד שכל הפרומיסים יסתיימו  דוגמה בהמשך |
| APP\_INITIALIZER - המשך  דוגמה לקונפיגורציה של ה APP\_INITIALIZER טוקן (מפתח DI) :  אדום- נייבא את הפונקציה בשם onAppInitFn מהמודול שבו היא רשומה (אני רשמתי אותה במודול של סרביס הקונפיג)    אדום- נוסיף למערך ה providers בתוך הדקורייטור @NgModule פרוביידר בשביל ה APP\_INITIALIZER. כחול- נגדיר את המפתח של הפרוביידר (ה DI Token) בשם APP\_INITIALIZER – זה חובה. ורוד- ורוד- נגדיר את הפונקציה onAppInit שכתבנו כפונקציה שיש להריץ בתהליך האתחול על ידי פרופרטי useFactory (useFactory אומר ל DI של אנגולר להריץ פונקציה במקום לאתחל אינסטנס של מחלקה) תכלת- לא יודעת מה זה. אולי כי ניתן לרשום הרבה פונקציות לאותו מפתח APP\_INITIALIZER אז רושמים multi:true. ירוק- deps הפונקציה מקבלת פרמטר מסוג AppConfigService, לכן נגדיר אותו כ dependency של הפונקציה בעזרת הפרופרטי deps. כך אנגולר יאתחל את האובייקט ויספק אותו לפונקציה.    דוגמה לפונקציה: כחול- נכתוב פונקציה, ורוד- ונשמור אותה בתוך משתנה בשם onAppInitFn . אדום- נעשה לה export. הפונקציה מקבלת פרמטר (תכלת) אובייקט מסוג AppConfigSerive, וקוראת למתודה loadConfig() של הסרביס (שחור). המתודה מחזירה אובייקט מסוג promise.    המשך דוגמה- בהמשך |
| APP\_INITIALIZER – המשך דוגמה  דוגמה לסרביס AppConfigService ולמתודה שלו:  אדום- הסרביס מקבל ב DI את ה Injector של אנגולר. למה? כי אנחנו צריכים את הסרביס HttpClient. אבל ה HttpClient זה סרביס שנטען ב DI, ובגלל שהסרביס שלנו AppConfigService מאותחל עוד בתהליך ה bootstrap אי אפשר לסמוך על ה DI כי הוא עוד לא מוכן, ואולי הסרביס HttpClient עוד לא אותחל בעצמו. לכן נשתמש ב Injector של ה DI וניגש דרכו ישירות לסרביס HttpClient בהמשך.  כחול- המתודה loadConfig(). שחור- נקבל את האינסטנס של הסרביס HttpClient מה Injector על ידי מתודה get() של האינג'קטור ונשמור במשתנה בשם http. תכלת- נעשה Http GET לקובץ קונפיג בשם app-config.json שנמצא על השרת. ורוד- נהפור את ה observable ל פרומיס. למה? כי ה APP\_INITIALIZER יודע לחכות רק ל פרומיס שיעשה resolve, הוא לא יודע לחכות ל Observable שיסתיים. ירוק- נירשם ל then של הפרומיס (הפונקציה שנעביר כפרמטר ל then() תופעל כשהפרומיס תסתיים בהצלחה (resolve), ונשמור את האובייקט שקיבלנו מקובץ הקונפיג במשתנה לוקאלי \_config מסוג AppConfig (אינטרפייס שכתבתי שמכיל את הפרופרטיס של האובייקט שב app-config.json). צהוב- נוסיף טיפול בשגיאות ונציג הודעת שגיאה למשתמש (אם הקונפיג לא נטען האפליקציה לא תרוץ טוב, צריך להודיע ליוזר). אם הפרומיס מחזירה שגיאה, נירשם על ידי catch() שמקבלת פונקציה שתופעל במקרה של שגיאה. הפונקציה מציגה alert.  ירוק כהה – האינטרפייס AppConfig. כרגע מכיל פרופרטי יחיד. |
|  |

XX

|  |
| --- |
| paths - נתיבים  "./" - מגדיר נתיב רלטיבי. לדוגמה: "./product.html" - מתייחס לקובץ product.html שנמצא באותו פולדר |
|  |
|  |
|  |
|  |

XX

|  |
| --- |
| הערות על JS |
| for … in … / for … of …  מה ההבדל בין הלולאות?  ב ES 2015 (JS החדשה) יש 2 לולאות for שונות: for … in … / for … of …  for … in …  עושה איטרציה על הפרופרטיס של אובייקט ומחזירה (נדמה לי) את שמות הפרופרטיס.  \*כאשר האובייקט הוא מערך- האינדקסים של המערך משמשים כפרופרטיס של המערך, ולכן- נקבל את אינקסים בחזרה  לדוגמה: prop מכיל את הערכים 0,1,2 (האינדקסים של המערך כי הם משמשים כפרופרטיס שלו)  let arr = ['aa', 'bb', 'cc'];  for prop in arr //prop = 0,1,2  //arr[prop]  for … of …  דומה ללולאת foreach - עושה איטרציה על אוספים, ומחזירה כל פריט באוסף |
| מודולים  מודול ב JS זה פשוט קובץ.  משתנים ופונקציות בקובץ מסוים זמינים רק באותו קובץ, ולא מוכרים מקבצים אחרים.  קוראים לזה מודולים של JS.  אם רוצים בכל זאת לגשת למשתנה / פונקציה מקובץ אחר - יש לעשות 2 דברים:  ייצוא של הפונקציה בקובץ המקור שלה (בעזרת exports נדמה לי)  ייבוא של הפונקציה בקובץ המטרה (בעזרת imports נדמה לי)  זה קל ומודולרי ואפשר לבנות קוד נוח וקריא בעזרת מודולים (קבצים) מרובים של JS. |
| קיצור של JS - איך להפוך סטרינג למספר?  עם אופרטור פלוס.  אופרטור פלוס על סטרינג - ממיר אותו לסטרינג (אני חושבת שזה יעבוד רק אם הסטרינג מכיל מספר)  דוגמה:  let str:string = '1';  let num:number = +str; |
|  |

XX

|  |
| --- |
| RxJs  איך להשתמש ב RXJS ב JS ונילה: פשוט לייבא את הסקריפט ל HTML מתוך CDN וכו'  <script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>  Observables עוזרים לנו לטפל במידע שמגיע בצורה אסינכרונית - כמו response מהשרת, כמו רצף של אירועים מאינפוט של יוזר (למשל לחיצה על עכבר / הקלדה בתיבת טקסט).  ב rxjs גרסה 6.x.x יש 2 פקג'ים עיקריים שמתוכם מייבאים את הפונקציונליות:  rxjs - מכילה את ה Observable, ומתודות ליצירת observables :    rxjs/operators - מכילה את האופרטורים:  import { map, filter, catchError, tap, take } from 'rxjs/operators';  אפשר להשתמש לדוגמה במתודה range() שבפקג' rxjs כדי לייצר observable sream (או פשוט observable) של מספרים (למשל מ-0 עד 10):    בקונבנציה - מוסיפים סיומת דולר $ לשם משתנה מסוג observable    אופרטורים משמשים לעשות מניפולציה על האירועים שה observable מרים (פליטת נתונים, שגיאה, סיום)  משתמשים במתודה pipe() של ה observable כדי להעביר את אירועי ה- observable דרך "צינור" של אופרטורים. ניתן להגדיר כמה אופרטורים שרוצים ב pipe(). בדוגמה- משתמשים באופרטורים map(), filter()  \*כאשר נפלט פריט - הוא עובר דרך הצינור של האופרטורים. לא מחכים עד שכל הערכים ייפלטו אלא הם עוברים בצינור אחד אחד ברגע שהם נפלטים.    נרשמים ל observable כדי לגרום לו לפלוט ערכים. Observable לא יפלוט נתונים עד שיש לו נרשמים לפלוט אליהם (הוא lazy) |
| rxjs - המשך - תיאוריה  RX = Reactive Extentions  תכנות "ריאקטיבי" - זה תכנות שפועל על עיקרון של תגובה למידע שמגיע אסינכרוני (וגם לשגיאות / סיום). כמו מידע שמגיע משרת backend למשל, או מאינפוט שיוזר מכניס, וכו'  rx זו הרחבה לשפה - שנותנת אובייקטים ומתודות לתכנות ריאקטיבי.  יש ספריות rx להרבה שפות תכנות- C#, JS וכו. RxJs - מרחיבה את JS.  Observable  ספריות Rx משתמשות בדבר שנקרא Observable (או Observable Sequence) - זה כמו מערך, אבל במקום לתת לי את כל המידע בבת אחת כמו מערך- הוא מספק רצף של מידע לאורך זמן. יכול לדוגמה לפלוט "a", אחרי שניה "b", אחרי עוד 2 שניות את "c". לכן הוא מכונה גם רצף מידע (sequence) או זרם (steam).  המידע מגיע אסינכרונית, כי לא "מקפיאים" את האפליקציה ומחכים לדוגמה שהיוזר יואיל בטובו לסיים להקליד, או שהתשובה תגיע סוף סוף מהשרת. בזמן שמחכים - עושים דברים אחרים. לכן זה אסינכרוני  ה observable מודיע לנרשמים על 3 דברים: פולט מידע, שגיאה, סיום.  כל אחד מה פליטת נתונים הזאת מכונים לפעמים אירועים של ה Observabel.  observable מתייחס לאירועים (של פליטת מידע, של שגיאה, של סיום) כאל אוסף.  Subscribe()  Observable הוא lazy – כלומר הוא לא יעשה את העבודה שלו (לא יפלוט נתונים) עד שנרשמים אליו בעזרת קריאה לפונקציה subscribe  לכן- יש להירשם אליו ברגע שרוצים להתחיל לקבל ממנו נתונים.  Subscribe() מקבלת עד 3 פרמטרים – כל פרמטר הוא פונקציה:  next function – מקבלת ומעבדת את הנתון הבא שהוא פולט. אם הוא פולט הרבה ערכים – ה next תיקרא עבור כל ערך שפולט.  error handling function – פונקציה שתופעל במצב של שגיאה  , וכל פרמטר מטפל במקרה שונה: נתונים שמגיעים, שגיאה, סיום.  complete function – פונקציה שתיקרא כשה observable סיים  Subscribe() – מחזירה subscription. זה אובייקט שאפשר להשתמש בו אם צריך למשל לעשות unsubscribe  unsubscribe() - ניתן לבטל observable על ידי unsubscribe    אופרטורים  Observables מאפשרים לנו לעשות מניפולציות על ה events שהם מרימים (פליטת נתונים, שגיאות, סיום)- בעזרת מתודות שנקראות אופרטורים.  כל אופרטור משנה את ה observable שהוא פועל עליו - בדרך מסוימת. ומחזיר observable חדש שבתורו מעיף את האירוע.  אופרטורים לא מחכים לכל הערכים שיגיעו ורק אז מעבדים אותם. הם מופעלים על כל event - ובכלל זה אירוע של פליטת מידע. כך שאם יש פליטת מידע 3 פעמים - האופרטור יופעל 3 פעמים (בכל פליטה של מידע)  אופרטורים לדוגמה: map, filter, take, merge, reduce  \*אופרטור דומה למערך בכך שהוא מחזיר הרבה ערכים. לכן- יש המון אופרטורים של מערך שפועלים על ה obserbvable בדומה למערך. לדוגמה: map() משנה את הערכים של ה observable. filter(), take() ועוד ועוד. יש פירוט בהמשך.  האופרטורים נקראים pipable כי ניתן לשרשר אותם. |
| rxjs - המשך  \*כדי לגשת לאופרטורים - צריך להפעיל את המתודה pipe() של Observable. כנראה שאי אפשר יותר לגשת לאופרטורים דרך ה observable עצמו ולשרשר אותם אחד לשני. צריך להשתמש במתודה pipe() ולהעביר אותם אליה אחד אחד כפרמטרים מופרדים בפסיקים.: אדום- מתודה pipe(), כחול- אופרטור catchError() (אם הוא יופעל - אחרי שיסתיים הטיפול בשגיאה (במקרה הזה כתיבה לקונסול)- הוא שולח שוב שגיאה עם טקסט "error has occurred" על ידי מתודת throwError() של rxjs שמה שהיא עושה- מחזירה אובזרבבל שלא פולט נתונים, רק מייצר שגיאה. לכן- השורה הבאה שתרוץ תהיה הוורודה    \*גם אנגולר משתמשת ב Observables בקוד הפנימי שלה. ה httpClient של אנגולר משתמש ב observables. |
| איך לייצר observable  of()  פונקציה של rxjs שמייצרת אובזרבבל שפולט מייד את הפרמטרים של הפונקציה (בדוגמה- יפלוט אחד אחרי השני 1, 2, 3      from()  כמו of() אבל עובדת עם מערך או אובייקט שהוא iterable או פרומיס או    empty()  מייצר אובזרבבל שמסתיים מיד, בלי לפלוט שום מידע. בדוגמה- רושמים לקונסול הודעת סיום.      throwError()  מייצר אובזרבבל שפולט מיד שגיאה. בדוגמה- רושמים לקונסול את השגיאה      interval() / timer()  מייצר אובזרבבל שפולט כל X זמן מספר עוקב, מתחיל מ אפס.  ימשיך לעבוד עד אינסוף (אולי אפשר לעצור אותו איכשהו? לא יודעת)  אדום- יפלוט כל שניה. אחת. כחול- הקוד שיבצע משהו כל שניה (כאן הוא סתם מוסיף את המספר למערך ואז נציג אותו על הדף (ירוק)  בדרך כלל נשתמש ב interval() כדי לעבד מידע כל כמה זמן. דוגמה- לעדכן שער מטבעות על ידי בקשת AJAX ל API  timer() - אם רוצים לשים קום-לג לפני פליטת האיבר הראשון - להשתמש ב timer() במקום interval  generate() - אם רוצים לעשות אינטרבל שמסתיים עם תנאי מסוים (למשל אחרי 10 פעמים)        דוגמה לשימוש ב interval() כדי לעדכן שער מטבע כל 10 שניות: זה מביא את שער המטבע מהשרת כל 10 שניות ומעדכן את המסך: אדום- ניצור אובזרבבל שפולט כל 10 שניות בעזרת Interval(). כחול- נשתמש באופרטור flatMap כדי לשנות את הערך שהאינטרבל פולט (מספרים עוקבים) לערך שחוזר מהשרת ב AJAX (למשל שער מטבע). ירוק- נקרא לפונקציה updateChart שמקבלת את שער המטבע ומעדכנת גרף.    אפשר פשוט לעשות new Observable() |
| איך לייצר observable - המשך  from() - משמש ליצור אובזרבבל מכל מיני דברים:  איך להפוך פרומיס promise לאובזרבבל    איך ליצור אובזרבבל של event listener מאובייקט jQuery  fromEvent()  איך ליצור אובזרבבל של event listener מאלמנט DOM (עובד גם עם אובייקט jQuery)    אפשר ליצור קונקשן של webSocket ולהשתמש בו כאובזרבבל (נדמה לי שזה subject) לקבלת המידע, וגם לשלוח דרכו מידע: |
| אופרטורים נפוצים  \*ב rxjs גרסה 6.x.x מייבאים את האופרטורים מהפקג' הבא:  import { map, filter, catchError, tap, take } from 'rxjs/operators';  \*כדי לגשת לאופרטורים - צריך להפעיל את המתודה pipe() של Observable. אי אפשר כנראה פשוט לגשת לאופרטורים דרך ה observable עצמו: אדום- מתודה pipe() - היא מקבלת כפרמטרים אופרטור אחד או יותר, והיא תדאג לקרוא לכל אחד מהאופרטורים האלה על ה events שה observable יורה (כלומר כשהוא פולט נתונים, זורק שגיאה או מודיע על סיום) |
| tap()  אופרטור שעוזר לדיבאג או לוגים. הוא מצוטט לסטרים ומאפשר לצפות בנתונים שהסטרים (ה observable) פולט - בלי לשנות אותם. דוגמה:    דוגמה לתוצאה של ה tap שנרשמה בקונסול: |
| catchError() אופרטור  תופס כל שגיאה שה observable מרים. חייב לזרוק שגיאה או להחזיר observable , לכן השתמשתי במתודה throwError(str) של rxjs - אשר מחזירה Observable שרק מחזיר שגיאה ולא פולט נתונים או עו  שה סיום. כלומר - אחרי שטיפלתי בשגיאה במסגרת ה catchError() - אני דואגת לשלוח שגיאה גם לנרשמים. |
| filter()  מפלטר את הנתונים שנפלטים. לדוגמה- יפלטר רק מספרים > 5 |
| map()  משנה את הנתונים הנפלטים. מקבל כפרמטר- פונקציה שעושה מניפולציה על הנתון ומחזירה במקומו נתון חדש. הפונקציה מקבלת את הנתון עצמו שנפלט. מחזירה any נדמה לי  בדוגמה - הנתונים הם מספרים (זה נתון שאני יודעת מראש) וה map() מכפיל כל נתון ב- 3 |
| flatMap()  לוקח מידע מאובזרבבל A, ומאחד אותו עם מידע מאובזרבבל פנימי B, וכש אובזרבבל B פולט מידע - הוא יחבר אותו למידע מאובזרבבל A |
| take()  מקבל מספר. מחזיר רק כמות של פריטים לפי המספר שקיבל. לדוגמה- זה יחזיר רק 5 נתונים מה observable ויפסיק, גם אם הגיעו אחר כך עוד |
| אופרטורים נוספים: cancel, retry, delay, Reduce (נדמה לי שאלו אופרטורים). זה בכל מקרה פעולות שניתן לעשות על observable |
| מתודה throwError("xxx")  זו מתודה שמייצרת observable שאינו פולט מידע כלל, הוא פולט רק שגיאה. זה טוב לטיפול בשגיאות - כאשר תופסים את השגיאה של ה observable המקורי, רושמים לוגים או מה שרוצים, ואז רוצים להודיע לנרשמים שקרתה שגיאה. משתמשים באובזררבלל הזה, ומה שהוא עושה זה פולט שגיאה לנרשמים.    \* ב rxjs 6.x.x - המתודה נמצאת בפקג': |
| אופרטור debounceTime  debounceTime(x) - מחכה שיעבור x זמן מאז שהגיע נתון ל stream ורק אם עבר הזמן - פולט אותו.  פולט רק את האיבר האחרון, בלי ההיסטוריה |
| אופרטור throttleTime  משמש לדוגמה לקליטה של אירועי תזוזת עכבר.  פולט אירוע, ואז מתעלם מהנתונים החדשים שמקבל במשך X זמן לפני שפולט את האירוע הבא |
| distinctUntilChange - אם מגיעים ערכים זהים- יסנן את הערך הכפול (האחרון לא ייפלט) |
| אופרטור bufferCount  ממתין שיגיעו X פריטים לאובזרבבל, ופולט אותם במערך בגודל X  דוגמה: אדום- שימוש באופרטור bufferCount עם גודל 3. כחול- מה שיודפס בקונסול (פולט מערך עם 3 האיברים הראשונים. אחר כך פולט מערך עם איבר יחיד כי האובזרבבל הסתיים. אם הוא לא היה מסתיים - היה מחכה שיגיעו עוד 2 פריטים ולא פולט מערך קטן מגודל 3. |
| איך לאחד כמה אובזרבבלים לאובזרבבל יחיד  merge() (פונקציה, לא אופרטור)  concat()  בדוגמה- ה merge יפלוט את הפריטים אחד אחרי השני לפי הסדר שהם מגיעים: 1 > 'a' > 2 > 3 |
| איך לחכות לכמה אובזרבבלים שיסתיימו כולם. ( כמו Promise.all() )  forkJoin() (פונקציה, לא אופרטור)  מאחד כמה אובזרבבלים שונים ומחכה שכולם יסתיימו, ומחזיר פריט יחיד - מערך עם כל הפריטים מכל האובזרבבלים.  משמש בעיקר לבקשות HTTP מרובות. כי הן מחזירות תשובה יחידה ומסתיימות. לא מתאים לאובזרבבל שמחזיר הרבה פריטים בסטרים, כי יחזיר רק את הפריט האחרון אחרי שהאובזרבבל יודיע שסיים.  \*אם אחד מהם לא מסיים, או יש שגיאה - ה forkJoin לעולם לא יחזיר תשובה! |
| subject ו- behaviorSubject  subject vs behaviourSubject  behaviourSubject – כשעושים לו subscribe() הוא מחזיר את הערך שקיים אצלו מפעם קודמת.  Subject – לא מחזיר ערך דיפולטי כשעושים לו subscribe(), רק כשמשתנה לו הערך.  איך ליצור Subject:  אדום- ניצור Subject חדש. כחול - נקרא למתודה asObservable() שלו כדי לחשוף אותו כאובזרבבל. זה יותר בטוח (ככה אי אפשר לקלקל את הסטרים של הסבגקט נדמה לי). ירוק- נוסיף מידע לסטרים. מי שרשום לאובזרבבל (סגול) מקבל את המידע  \*מידע של הסטרים שמגיע לפני ההרשמה הולך לאיבוד. הנרשם יקבל רק מידע חדש שמגיע. כדי לפתור את זה משתמשים ב behaviorSubject      BehaviourSubject  1. מקבל ערך דיפולטי בקונסטרקטור  2. הנרשם מקבל מהאובזרבבל את הערך האחרון של ה subject מיד ברגע ההרשמה.  דוגמה- אדום- ניצור BehaviourSubject חדש. הוא מקבל ערך דיפולטי 99 (כחול). ירוק- הסבג'קט מוסיף את 98 לסטרים. ורוד- נרשמים לאובזרבבל של הסבג'קט ומקבלים את הערך האחרון מיד (98). \*אם השורה הירוקה לא היתה קיימת- היינו מקבלים 99 |

XX

|  |
| --- |
| VS CODE – הערות (VSCODE)  אם רוצים לייבא אוטו' את ה import statements ב TS – יש להוסיף פלאגין (extention):  ללחוץ על הסמל הזה בצד שמאל למטה ולרשום בחיפוש: auto import . להתקין ולעשות ריסטרט ל VS CODE      קיצורים:  Ctrl+backtick – פותח / סוגר טרמינל. כדאי לפתוח ולהשתמש ב angular cli כדי ליצר קומפוננטות וכו'.  F2 – נדמה לי שזה עושה ריפקטור לשמות באנגולר. אבל לא בטמפלייט – בטמפלייט צריך לשנות ידנית.  נקודהCtrl+ - כדי לבחור import ב TS |
|  |

XX

|  |
| --- |
| הערות על package.json  הפקודה npm install מסתכלת על הקובץ package.json ומתקינה את ה packages שמוגדרות בו.  הערה לגבי גרסאות של package – לכל גרסה יש package ו- npm יתקין את הגרסה הנכונה.  כל גרסה מורכבת מ מספר ראשי, משני, מספר patch.  ^ - מקבע את הגרסה הראשית. לדוגמה ^3.1.1 – מאפשר להתקין את 3.2…, 3.9… וכו'. לא מאפשר להתקין את 4.x.x  ~ - מקבע את הגרסה המשנית. לדוגמה ~3.1.1 – מאפשר להתקין עד 3.1.9 אבל לא מאפשר להתקין את 3.2.x  דוגמאות: |
|  |
|  |

XX

|  |
| --- |
| SASS (Syntactically awesome style sheets)  אנגולר CLI משתמש ב SASS ככלי דיפולטי לעשות pre-processing לקבצי CSS.  SASS הוא CSS Pre-Processor  זה כלי שמעבד את ה CSS לפני הקומפילציה של האפליקציה ועושה לו transpile ל CSS תקני שהדפדפן יודע לקרוא.  יש כמה כלים כאלה - SASS, LESS, Stylus. הם די דומים אחד לשני.  משתנים $  אפשר להגדיר משתנים ולתת להם ערך, ואז להשתמש במשתנים האלה בחוקי ה CSS.  שם משתנה חייב להתחיל בדולר $  דוגמה: (יש לעשות ng serve כדי לראות את השינויים בדפדפן)    ניתן לייבא קובץ SCSS אחד לתוך קובץ SCSS אחר בעזרת הכרזת @import כמו ב CSS רגיל: SASS ישרשר את הקובץ המיובא לקובץ המייבא - וייצור לנו קובץ אחד גדול.    mixins  זה אוסף של פרופרטיס של CSS שמוגדרים ביחד, ואפשר אחר כך לכלול אותם בחוק CSS .  דוגמה: אדום- נכריז על מיקסין בעזרת @mixin (צהוב) ושם המיקסין (תכלת) עם פרמטר בשם $border-radius מוגדר בתוך סוגריים (ירוק). נגדיר את הפרופרטיס בתוך המיקסין. ירוק בהיר- הפרמטר $border-radius משמש להגדרת הערך של border-radius.  לבן- נשתמש במיקסין בתוך חוק CSS רגיל של מח' box - בעזרת @include (ורוד) ושם המיקסין (תכלת) עם סוגריים (ירוק) כמו שקוראים לפונקציה  כעת המח' box כוללת בתוכה פרופרטיס של border, border-radius, padding כפי שהוגדר במיקסין.    הערה: אם לא רוצים פרמטרים - אפשר להשמיט את הסוגריים מהגדרת המיקסין, אבל ב @include יש לקרוא עם סוגריים ריקים.  פונקציות  דומה ל mixin אבל פונקציה יכולה להחזיר ערך  יש עוד דברים ב SASS כמו פונקציות ולא יודעת מה עוד |
| איך להוסיף SASS לאפליקציית אנגולר   1. נוסיף פלדר בשם sass 2. נוסיף מתחתיו 3 קבצים:   variables.scss  mixins.scss  styles.scss   1. נייבא את קבצי variables, mixins אל תוך קובץ styles.scss :      1. נקנפג את קובץ angular.json – נוסיף ל styles את הקובץ sass/styles.scss שיצרנו. (הנתיב רלטיבי לפולדר src או שלא)      1. בקומפוננטות: ניצור קובץ scss לקומפו. נייבא לתוכו למשל את ה קובץ variables : כך נוכל להשתמש במשתנים של ה SASS בתוך קובץ ה scss של הקומפו      1. נגדיר אותו בדקורייטור @Component של הקומפו:     זהו! ה CLI משתמש ב SASS באופן דיפולטי |
| איך לעבוד עם מעצב:  דרך קלה מאוד לעבוד עם מעצב זה דרך תוכנת zeplin. אפשר להיכנס לאתר שלהם, או להוריד תוכנה לדסקטופ. לי הספיק האתר.  אם המעצב עובד עם תוכנת zeplin זה מעולה – הוא מעלה את המסכים שלו לתוכנה, והמפתח יכול לראות את המסכים ובכל מסך – לבחור אלמנטים ולראות את ה CSS שלהם. מאוד כייף וקל לעבוד  \*הייתי צריכה לבקש קבצי פונטים וגם תמונות חתוכות מהמעצב, הן לא הגיעו ב zeplin כמו שצריך |

XX

|  |
| --- |
| Angular Material  זו ספריה שמכילה:  קומפוננטות UI איכותיות שניתן להשתמש בהן באפליקציות.  סדרה של מחלקות css לעיצוב של טקסט (זה נקרא typography)  (הקומפו' בנויות באנגולר וב TS, תומכות בשפות שונות ואולי גם ב RTL, יש להן API פשוט, עברו בדיקות, ניתן לקסטם אותן, הן מהירות, יש תיעוד אבל לא 100%)  דוגמאות לקומפוננטות של אנגולר מטריאל - כפתורים, צ'קבוקס ורדיו, מודל, ספינרים, אייקונים, ועוד.  כל קומפו כתובה במודול נפרד בתוך ספריית המטריאל. לכן- עבור כל קומפו שבוחרים להשתמש באפליקציה - צריך לייבא את המודול שלה.  באתר של מטריאל (material.angular.io) יש רשימת כל הקומפו' ולכל קומפו' יש תיאור של ה API שלה, דוגמאות, איזה מודול יש לייבא וכו'.  מטריאל מגיעה עם כמה themes (theme זה סטים של צבעים).  ניתן לבחור אחד מה themes או ליצור אחד חדש משלי.  ניתן גם לקסטם את ה typography.  angular material vs bootstrap  אנגולר מטריאל ו bootstrap הן 2 ספריות שונות שנכתבו בידי צוותים שונים.  אמנם bootstrap יותר ותיק ומכיל יותר רכיבים, אבל מטריאל מתאים לאנגולר ואפשר להשתמש בו בקלות, בעוד bootstrap עובד עם JS וצריך לפעמים לעשות כל מיני מניפולציות כדי לעבוד איתו.  לכן אם מטריאל מספק את הצרכים (אפליקציה עם UI פשוט) - עדיף להשתמש במטריאל |
| Angular Material - getting started:  יש תיעוד על איך להוסיף אנגולר מטריאל לפרוייקט - באתר של אנגולר > ריסורסים > מטריאל  1. נוסיף לפרוייקט את הפקג'ים הבאים דרך npm install --save  @angular/cdk - "ערכת פיתוח של קומפו' " (component development kit) - זו ספריה שמאפשרת לבנות קומפו' ל WEB בלי המראה של material design. משמש לדוגמה לבנות טבלה reusable שלא נראית כמו material design וגם את הפקג' של אנגולר מטריאל עצמה  @angular/animations - אם רוצים אנימציות (תנועה על המסך) צריך להתקין את הפקג' (לא יודעת אם הוא חובה)  ammer.js - ספרייה שמטפלת בפעולות של היוזר: נגיעה של האצבע, swipe, וכל מיני תנועות שהיוזר עושה עם האצבע (הפקג' הזה לא חובה אבל מומלץ לעבודה יותר טובה עם מסכי מגע)  2. נוסיף theme - theme זה אוסף צבעים (מבחינת עיצוב. צבעים וכו').  אפשר פשוט לייבא את הקובץ של ה theme לתוך הקובץ styles.css הראשי (ואולי אפשר לייבא דרך styles שבקובץ הקונפיגורציה angular.json )  ה themes נמצאים בנתיב:  node\_modules/@angular/material/prebuilt-themes  דוגמה: (צריך להשלים את שם הקובץ indigo-pink.css ל @inport    הערה: הטילדה בתחילת הסטרינג תגרום ל webpack לקחת את הקובץ מספריית node\_modules. זה גורם לנתיב להיות יחסי ביחס לפולדר node\_modules  3. נייבא את המודול לתוך ה app.module שלנו:  אם רוצים להוסיף אנימציה יש לייבא את מודול BrowserAnimationsModule  אם לא צריכים אנימציה אפשר להסתפק במודול NoopAnimationsModule (נמצא באותה ספריה)  import {BrowserAnimationsModule} from '@angular/platform-browser/animations';  @NgModule({  imports: [BrowserAnimationsModule],  })  4. נחליט על קומפו שרוצים להשתמש בה - ולכל קומפו נחזור על הצעד הזה:  נייבא את המודול של הקומפו' למודול שלי  הערה: כל קומפו מוגדרת במודול נפרד. איך יודעים מה המודול של הקומפו? מסתכלים בטאב api של הקומפו באתר של אנגולר מטריאל  הערה: אם משתמשים בהרבה קומפו' של מטריאל - זה יפוצץ לי את המודול שלי באימפורטים. זה לא נוח לקרוא ולא נוח לתחזק. עדיף ליצור מודול יחיד בשם MdComponentsModule למשל, ולעשות ממנו exports של כל המודולים של הקומפו של מטריאל, ואז- כשנייבא אותו אל המודול X שצריך את הקומפו של מטריאל - אז מודול X יקבל את המודולים של מטריאל ממנו. כד נבודד את המודולים של MD לתוכו והם לא יתפרסו לי על פני כל המודול שלי. (הערה- לקרוא שוב על Imports/exports של מודול ועל מודולים בכלל באנגולר. לא צריך לעשות import אם רק רוצים לעשות export! לא זוכרת את זה בדיוק- לקרוא על זה) |
| typography  אפשר גם לעצב יפה את הטקסט של האתר בעזרת מטריאל.  יש כל מיני css classes שאפשר להתשמש בהן. כל המחלקות יש להן תחילית של mat-xxx  אפשר להוסיף את ה css classes על אלמנטים שונים בדף ה HTML, לדוגמה:    או שאפשר פשוט להגדיר את המחלקה mat-typography על האלמנט <body> וזה יחלחל לכל האלמנטים שבדף.    רשימת ה css classes: (העתקתי מהאתר של מטריאל. כדאי להסתכל שם למידע עדכני. כל מח' מקבלת קידומת mat-\* לפני שמה)    ייבוא הפונט של מטריאל: מטריאל עובדת עם פונט שנקרא roboto . אפשר כמובן לשנות ולעבוד עם איזה פונט שבא לי.  כדי לעבוד עם פונט roboto - צריך לייבא אותו עם @import לקובץ ה CSS שלי. (יש באתר של מטריאל את ה URL של הפונט - פשוט להעתיק את ה URL לתוך ה @import (לא צריך font-face או משהו כזה)    הערה: אפשר לראות הסברים באתר של מטריאל > guides > using angular material typograpy |
| איך לקסטם את ה typography  יש הוראות באתר של מטריאל.  לדוגמה- נחליף את הפונט בפונט של גוגל חינמי בשם open sans (פונט יפה ופופולרי ומודרני)  נלך ל fonts.google.com ונחפש שם את הפונט לפי השם open sans ונעתיק את ה URL בשביל ה @import. בשביל ביצועים טובים יותר - לא כדאי לקחת את כל ה font family רק חלק קטן ממנה. יש באתר דוגמאות והוראות איך לכלול את הפונט באפליקציה שלי.  נייבא את הפונט לתוך קובץ ה SASS שלי בעזרת @import וה URL שהעתקתי מגוגל    נגדיר את הטייפוגרפי בקובץ SASS :  1. נגדיר  דוגמה: ירוק- ניצור אובייקט חדש של הגדרות טייפוגרפיה בעזרת הפונקציה בשם mat-typography-cofig() (ורוד). נשמור אותו בתוך משתנה בשם $app-typography (אדום). תכלת - נעביר לפונקציה mat-typography-cofig() פרמטר בשם $font-family שמכיל את שמות הפונטים החדשים. צהוב- נעביר לפונקציה פרמטר בשם $headline שמכיל הגדרות שונות מהדיפולט עבור גודל הפונט, גובה השורה, והדגשה (BOLD ). זה עבור הגדרות של כותרות.  לבן- נעשה @include ל mixin בשם angular-material-typoghraphy שמקבל כפרמטר את המשתנה $app-typography שיצרנו.    הערות:  אפשר לפתוח את קובץ \_theming של מטריאל ולהסתכל בהגדרה של הפונקציה mat-typography-config() היא מקבלת כל מיני פרמטרים- את כל ה css classes של ה טייפוגרפי. לכולן יש ערכים דיפולטיים, ואם רוצים לקסטם אותם צריך להעביר לפונקציה הזאת את הפרמטרים המתאימים. |
| theme  זה אוסף צבעים.  primary  accent(מודגש. לכפתורים למשל)  warn  אפשר לבחור בין ה themes המובנים של מטריאל, או ליצור custom theme משלי |
| איך ליצור custom theme  יש הוראות באתר של מטריאל איך ליצור custome theme. אני מפרטת את מה שכתוב שם:  נבחר צבע+גוון לכל אחד מהצבעים של מטריאל: דיפולט, מודגש (לכפתורים וכו') ואזהרה.  אפשר ללכת ל ל color tool של מטריאל (material.io/color) ולבחור שם את הצבעים ולכל צבע את הגוון שלו. יש טבלה.  1. נוסיף קובץ SASS לפולדר SRC (אפשר לקרוא לו theme.scss) (scss זה הסיומת של קבצי SASS. ה CLI אוטו' יעשה לו transpile ל CSS)  2. נייבא לקובץ SASS שיצרנו את קובץ ה theming של מטריאל בעזרת @import: (הטילדה ב @import של CSS מצביעה על נתיב יחסי לפולדר node\_modules) (@import זה CSS rule שמייבא קובץ CSS אחר לתוך הקובץ שלנו) (אין צורך להוסיף סיומת scss היא דיפולטית אם ה @import מופיע בתוך קובץ scss)    2. נקנפג את ה CLI של אנגולר לכלול את הקובץ SASS בסטיילים של האפליקציה: בקובץ ה angular.json בחלק styles נוסיף את הקובץ שלנו למערך הקבצים:    3. נכלול בקובץ SASS שלי את ה core styles שמוגדרים ב מטריאל. ה styles האלה נותנים למטריאל מראה אחיד. הם מוגדרים ב mixin שנקרא mat-core. נכלול אותו בקובץ SASS שלי בעזרת @include של SASS:    4. נגדיר את ה theme :   1. נגדיר 3 צבעים שישמשו אותנו ב theme - צבע עיקרי, צבע מודגש (לכפתורים וכו'), צבע אזהרה. 2. ניצור theme חדש מהצבעים שהגדרנו ונשמור אותו במשתנה. 3. נוסיף את ה mixin עם ה theme החדש שיצרנו   דוגמה: ירוק- ניצור 3 אובייקטים של צבע בעזרת פונקציה mat-palette() (אדום) ונשמור אותם במשתנים. הפונקציה  mat-palette() מחזירה אובייקט צבע שיש לו צבע+גוונים. היא מקבלת 2 משתנים: הצבע (צהוב), גוון (תכלת). הצבע הוא לא סטרינג פשוט אלא אובייקט מורכב שמכיל צבע+גוונים שונים.  ורוד- ניצור אובייקט theme בעזרת הפונקציה mat-light-them() ונשמור אותו במשתנה בשם $app-theme. הפונקציה מקבלת את 3 המשתנים של הצבעים שיצרתי.  לבן- נוסיף בעזרת @include את ה mixin בשם angular-material-theme שמקבל פרמטר - את ה theme שיצרנו.    הערות והסברים:   * כדי לבחור את הצבעים - צריך להיכנס לאתר של מטריאל אל הכלי שנקרא color tool. הוא מראה את כל הצבעים האופציונליים, ולכל צבע- את כל הגוונים (בהיר עד כהה). אפשר לשחק שם עם הצבעים והגוונים עד שנמצא שילוב טוב ואז להשתמש בו. בדוגמה למעלה - השתמשנו ב כחול גוון 600 בתור צבע עיקרי, צהוב גוון 700 לכפתורים והדגשות, ואדום גוון 500 (דיפולטי) לאזהרה * ניתן לפתוח את קובץ \_theming של אנגולר ולראות את הגדרות הפונקציות השונות שהשתמשתי בהן כמו mat-light-theme וכו'. לכל פונקציה יש פרמטרים חובה ודיפולטים, אם נסתכל על הקוד זה יתן עוד הבנה (למשל לפונקציה mat-light-theme לא חייבים לתת 3 פרמטרים, אם משמיטים את ה warn היא תשתמש בדיפולט - אדום) * הערה בקשר ל $mat-blue וכו' - אלו משתנים מקובץ \_theming שמייצגים את הצבעים השונים של מטריאל. כל משתנה כזה מכיל צבע +גוונים. יש משתנה לכל צבע, שמות המשתנים מתחילים ב $mat-xxxColorName * הפונקציה mat-palette() מחזירה אובייקט שמחזיק צבע + גוונים. היא מקבלת פרמטר חובה - צבע (אובייקט שיש לו צבע+גוונים, ולא סטרינג עם שם הצבע). פרמטרים רשות- הגוונים עבור דיפולט, כהה, בהיר) * צבע ה- warn בדיפולט הוא אדום ואפשר להשמיט אותו אם רוצים להשתמש בדיפולט של מטריאל. * המשתנים הם לוקאליים שלי - אני יכולה לקרוא להם באיזה שם שאני רוצה |

|  |  |  |
| --- | --- | --- |
| איך לדבג אפליקציית אנגולר ממחשב אחר / מטלפון נייד   1. נעלה את שרת הפיתוח של ה CLI עם IP 0.0.0.0   ng serve –host 0.0.0.0  כעת השרת פיתוח יאפשר לגשת לאפליקצייה עם ה IP של התחנה משאר המחשבים שברשת (אם הנייד גולש ב wifi לאותה רשת של המשרד – הוא יוכל גם כן לגשת)   1. נפתח את ה IIS EXPRESS לגישה מהרשת- כדי שישרת בקשות של ה API  **Step 1 : Configure IIS Express** For Visual Studio 2015,  open file ***<solution>/.vs/config/applicationhost.config*** and ***localhost*** in **‘bindingInformation’** to **‘\*’.**After making changes, it should like below  [HTTP Binding](https://www.lakshmikanth.com/wp-content/uploads/2015/10/HTTP_Bindings.png) **Step 2: Allow URL Access**  * Open command prompt in ‘Administrator’ mode * Run following command      |  |  | | --- | --- | | 1 | netsh http add urlacl url=http://\*:51603/ user=everyone |   **Note:** Run this command with your IIS listening port. In my case above, application was listening on port 51603 **Step 3: Configure Firewall Port**  * Go to the “Control Panel\Windows Firewall” * Click “Advanced settings” * Select “Inbound Rules” * Click on “New Rule …” button * Select “Port”, click “Next * Fill your IIS Express listening port number, click “Next”   [Inbound Rule](https://www.lakshmikanth.com/wp-content/uploads/2015/10/Inbound_Rule.png)   * Select “Allow the connection”, click “Next” * Check where you would like allow connection to IIS Express (Domain, Private, Public), click “Next” * Enter rule name and click “Finish” |