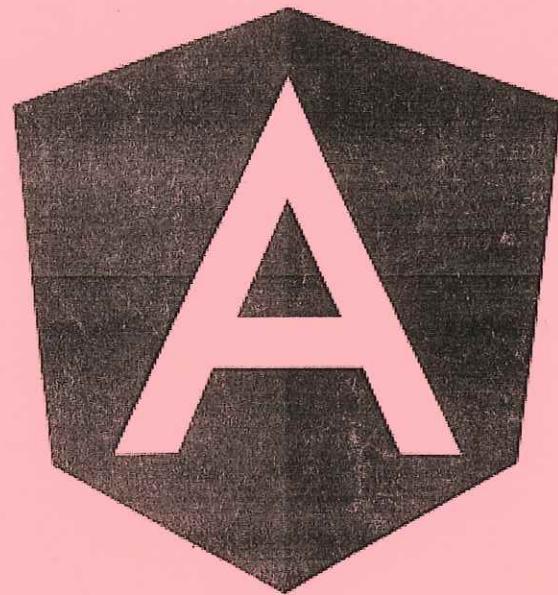


מג'זין
טלפון 052-7142050

בג"ד



Angular

גירסה 7

מהדורה רביעית (12/2018)

© כל הזכויות שמורות לפנינה אורן

אין לקרוא, להדפיס, לצלם, להעתיק, להעביר או לעשות כל שימוש
בחומר זה ללא רשות כתובה ומפורשת

pnoeren1@gmail.com 052-7142050

תוכן עניינים

4.....	הקדמה
5.....	הארQUITטורה של angular
7.....	יצירת פרויקט אנגולר
8.....	הסבר כללי על מבנה הפרויקט ועל הקבצים שהוא מכיל
10.....	הרצה הפרויקט
10.....	יצירת רכיבים חדשים Angular cli
12.....	מודולים / מודול ראשי - Root Module
12.....	AppModule - RootModule
14.....	Lazy Loading
15.....	- קומפוננטות Components
15.....	metadata העיקריים בקומפוננטה
18.....	Data binding
20.....	יצירת מחלקות נתונים
21.....	שלושת השלבים יצירת קומפוננטה
22.....	אירועים
25.....	Input
26.....	Output
28.....	טפסים
28.....	[(ngModel)]
30.....	ולידציות
33.....	Services
35.....	Promise
37.....	Http Client

Angular v.7

37.....	Observable
38.....	get / post
40.....	Async pipe
42.....	Observables operators
43.....	Routing – Navigation
44.....	נוווט ע"י קוד
45.....	נוווט באמצעות פרמטרים
47.....	LifeCycle Hooks
48.....	תקשרות בין קומפוננטות Component Interaction
48.....	Input – Output
49.....	Template Reference Variable
49.....	Service
49.....	ViewChild – ContentChild
54.....	Directives
54.....	מודולים Directives
56.....	Custom directive
59.....	Pipes
59.....	מודולים Pipes
60.....	Custom pipes
61.....	Lazy Loading
61.....	טעינה ע"י routing
64.....	טעינה ע"י קוד
66.....	Change Detection
68.....	חידושים ועדכוניםanganolr 6 7

הקדמה

עלם הפיתוח עבר שינויים רבים, ובמיוחד בשנים האחרונות בהן השימושים הם מוכרים מן המציאות עצמה. עלם הפיתוח חיב להתקנים את עצמו לשימושים השונים של הלוקחות, ליעודים השונים של המערכת, לצרכים המשתנים ולקצב התקדמות הטכנולוגיה בכלל.

התפתחות המרשימה ביותר היא דואק באצד לקוב, וזאת מכמה סיבות הקשורות אחת בשניה. השימוש המקורי מתעצם מיום ליום וממילא יש צורך למצאו פיתרון או תחליף לעובדה באצד שרת, מכיוון של צרכני האפליקציה פונים לאוטו שרת שעשו את העבודה והעומס עליו הוא לא העומס שהיה פעם בזמן התפתחות הטכנולוגיות צד שרת הקיימות ויש למצאו דרך לנטרל כל פעילות שאינה חייבה להתבצע דואק בשרת. כל פעילות כזו מומלץ להעבירה לצד לקוב.

בנוסף, המחשבים של היום הרבה יותר חזקים מאי פעם, מAMILIA ניתן לבצע הרבה יותר פעולות מצד לקוב.

בנוסף לכך, העבודה באצד לקוב מהירה הרבה יותר מאשר העבודה באצד שרת, ואחד הנושאים החשובים ביותר היום זה מהירות ביצועים. טכנולוגיות הצד לקוב יכולות לתת מענה יותר טוב ויעיל לבעה זו.

אנגולר זהה טכנולוגיה חדשה המאפשרת פיתוח מלא מקצה ועדת באמצעות העבודה באצד לקוב. ככלומר, אングולר אספה התנהגויות שונות שכטבנו עד היום באצד שרת ואייפשרה אותן באצד לקוב, כמו למשל `bind` למודול לתצוגה, שירותים והזרקות, מודולים ושכבות. אングולר בעצם באה ואומרת שניתן לפתח אפליקציה שלמה באצד לקוב בשונה مما היה עד היום שהשימוש באצד לקוב רק פעולה על הדף עצמו ולא הינו כל הטכנולוגיה שבסביבו. הטכנולוגיה לדוגמה שהיא סיבוב דף אינטראנט הינה `asp.net webforms` או `mvc` וככ'.

אחד העקרונות החשובים באנגולר זה העצם העבודה עם `option` - דף `html` אחד בלבד שאיתו יש לעבוד לאורך כל הדרך. דף זה עולה פעם ראשונה וזהו. אין יותר תחלופה של דפי `html`.

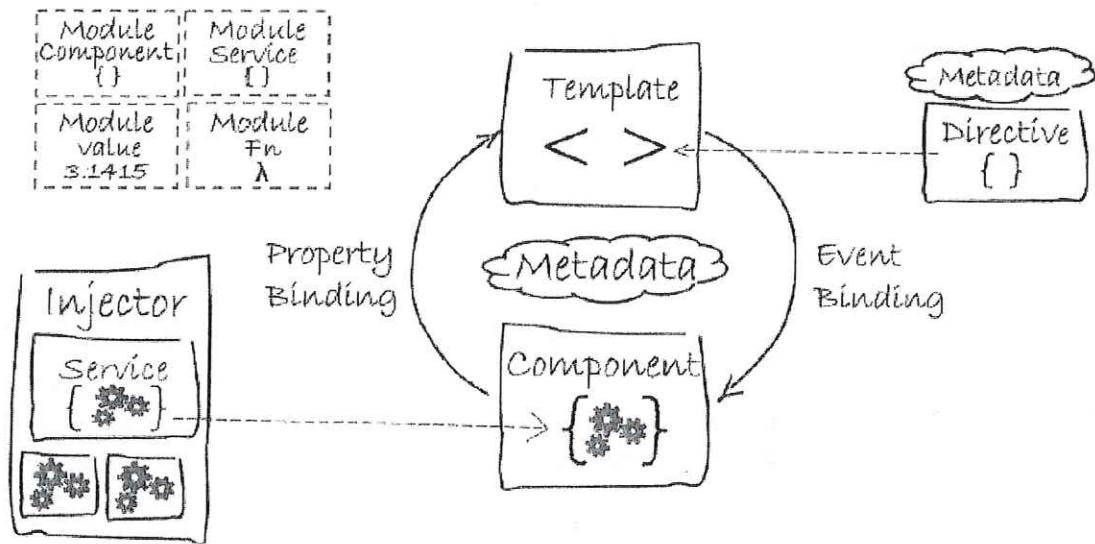
כאשר מנסים ברמתה זו את התצוגה בפועל של ה `html` אל המש坦ש (שנקרא DOM) זה נעשה באופן מאד מהיר, בהזתק של זמן שהמש坦ש אינו מרגיש כלל. מאד יידידותי, בדומה מאוד להחלפת מסכים במובייל.

אנגולר היא אינה שפה או ספרייה, היא טכנולוגיה, וכך היא מכתיבה את צורת העבודה.

אנגולר (של גוגל) נכתבה בשפת `typescript` (של מיקרוסופט). שפה זו היא שפה חדשה הדומה לתקן החדש של שפת `javascript` `es6` או `es7`.

שפת `javascript` בעבר תהיל'ך שימושי המקדם ומשפר מאד את העבודה משפה שהיא `typeless` לשפה עם `type` מחלקות ירושות ממשקימים ועוד ועוד – שפה לכל דבר הבניה כמו שפות מתיקדות אחרות המוכרות לנו עד היום משפטן צד שרת. שפה זו אינה נתמכת בכל הדפנסים וכן ישנו מנוע הדואג לקמפל את הקוד ולהמירו לשפת `javascript` בהתאם הנדרש.

הארכיטקטורה של angular



הארכיטקטורה של אングולר בנויה מכמה חלקים עיקריים:

Modules

אפליקציות שבונים עם אングולר הין אפליקציות מודולריות. מודול זה בעצם כולל כל בתוכו ניהול הפעולות והתנהגוויות שונות עבור נושא מסוים. כמו למשל: מודול ניהול עובדים, מודול זה יכול בתוכו מרכיבים שונים כמו הוספה/עריכת עובד, צפיה במספר הפניות שלו וכו'.

בכל אפליקציה מודולרית יהיה לנו לפחות מודול אחד, מודול הבסיס `root module` שייקרא `.AppModule`.

במערכות קטנות זה יהיה המודול הבסיסי והיחיד שייכיל את כל האפליקציה, במערכות מורכבות יותר, יהיו לנו הרבה מודולים קטנים הנקראים `feature modules`.

Components

צורת הפיתוח באングולר נכתבת כרכיבי תצוגה הכוללים ביחידת אחת שני חלקים. היחידה עצמה נקראת קומפוננטה והוא מורכבת משני חלקים – לוגי ויזואלי. חלק אחד שלו זה החלק הלוגי שהוא המחלקה שניצור עבורך. המחלקה – זהו בעצם קוד `ts` המגדיר `class` ובו מאפיינים ומетодות. החלק השני שלו זה החלק היזואלי שהוא `html` שניצר לשםך. את מלאכת החיבור בין החלק היזואלי לחלק הלוגי אングולר יבצע.

לדוגמא, רכיב רשימת תלמידים יוכל מחלקה ובה מאפיין `students` – מערך מסווג עובד, יוכל מתודת `selectStudent()` שתציב לתוך המאפיין `currentStudent` את העובד שיבחר בלחיצת עכבר מן הרשימה. הרשימה עצמה תוצג באנטם כשלחו עם אפשרות בחירת תלמיד מתוך הרשימה.

DataBinding

המנוע שמחבר לנו בעצם בין הנתונים לבין התצוגה, מבל' לעשות את זה ידנית.
bind זה יכול להיות לכמה Gründe:

Input: קליטת מידע פנימה לקומפוננטה,

Output: פליטת מידע מהקומפוננטה החוצה

Two-way-binding: קליטה ופליטה גם יחד.

Directives

זהוי בעצם מחלוקת שהוצאה כ Directive @Directive ותפקידו להתריער ולסייע בחלול התצוגה.

לדוגמה, מחלוקת הקשורות לבנייה של תוכן ה html, הן ישירות תוכן / יחוzu על תוכן וככ' בתוך ה template, כמו למשל `ngFor`* או `ngIf`*

מחלקות הקשורות לתוכן, כמו למשל attributes של Model המ楊וס את המאפיין במודול לשדה הקלט.

ניתן גם לכתוב באופן עצמאי directives ולהוסיף על הקיימים.

Services

זהוי בעצם מחלוקת המיעודת לתת לנו שירותים, אין תחביר מיוחד למחלוקת זו, וניתן לבנות אותה איך שרצים, הרעיון הוא שחלוקת זו מרכזת את כל השירותים הנדרדים, וכשנגיד לו לקומפוננטה מסוימת שתצרוך את המחלוקת זו – נקבל אותה בהזרקה לתוכה `constructor`s ואז יוכל להשתמש בה שימוש בחלוקת. פירוט בהמשך.

Dependency injection

זהוי בעצם יכולת לייצר מופע למחלוקת שאנו תלויים בה כאשר אנו מייצרים את המחלוקת הצורכת.

השימוש הנרחב בתוכונה זו הוא בשירותים.

שירות שאנו מגדירים כ provider יוכר בכל המופע אליו הוא יוזרך. ניתן להגיד שירותים במודול עצמו ועוד זה יוכר בכל הקומפוננטות, ניתן גם להגיד עבור קומפוננטה מסוימת שירות מסוים וזה יוכר בתוך אותה קומפוננטה.

ישנם עוד מרכיבים לאפליקציית אנגולר, אולם נתמקד בהם בהמשך.

יצירת פרויקט אングולר

לצורך עבודה עם אングולר יש להתקין קודם כל המחשב עצמו את `node.js` (התקנה דרך האתר <https://nodejs.org/en/download/>).
node.js זהו מנוע שמודע להרץ javascript על המחשב.

לאחר מכן, נרצה להתקין ספריות שונות מבוססות על אותו מנוע, כמו למשל אングולר, לצורך כך, ישנו כל המנהל את כל הספריות הללו, כדי זה נקרא `npm`. (מנהל ספריות `js`)
כל ספריה ניתן דרך הכלי זהה. שימוש בכלי זהה מאפשר רק לאחר התקינה של `node.js`.

לצורך עבודה קלה ונוחה עם אングולר – נוצרCLI בשם `angular` כדי זה מאפשר הרצת פקודות ב command line ובכך יוצר קבצים/קודים באופן אוטומטי ונוסח הרבה מאד זמן עבודה.
כל זה מיועד גם עבור build לסייעות שונות, בניית טיטים לאפליקציה ועוד.

התקנת cli angular תבצע ע"י הרצת פקודה command:

```
npm install -g @angular/cli
```

פקודה זו מבקשת התקינה של ספרייה cli angular, הפרמטר `-g` – מצין שההתקינה תבוצע גלובלית על המחשב ולא על התיקייה בה command פתוח כרגע.

לאחר שהתקן cli ניתן להשתמש בכל הפקודות שהכלי זהה יצר עבור אングולר.
כל הפקודות תהיה תחת השם `ng`.

הפקודה הראשונה תהיה `new ng` – פקודה זו תיצור פרויקט אングולר בסיסי, המכיל את כל המנעדת הנוצרת, מודול ראשי וקומפוננטה ראשית.

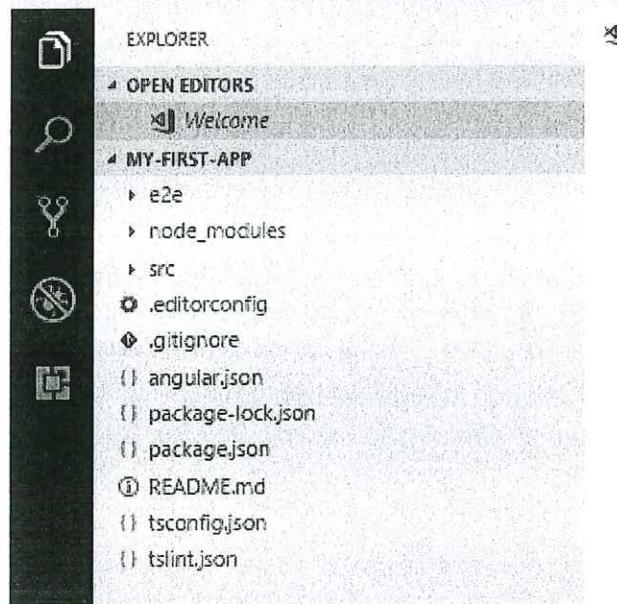
לפקודה זו יש לתת את שם הפרויקט אותו רוצים ליצור, לדוגמה: (`my-first-app` זהו שם הפרויקט)

```
ng new my-first-app
```

יש לשים לב שהפקודה תפעיל את עצמה במיקום שבו ה cmd עומד. במידה ורוצים מיקום שונה
למקם את ה cmd בתיקייה הרצויה.
פקודה זו תשאל שאלות כדי ליצור את הפרויקט לפי העדפות הרצויה – יש לסמן את התשובות
לפי התהילה.

לאחר שהרצת פקודה זו תסתיים, נקבל פרויקט בסיס מלא ועובד.
ישנם כלים רבים לעבודה עם צד לקוח, בחופרת זו נעשה שימוש ה `visual studio code` שהוא חינמי
וש לו הרבה מאד הרכבות חינמיות.
אם נפתח את הפרויקט נראה אותו כך:

Welcome - my-first-app - Visual Studio Code



הסבר כללי על מבנה הפרויקט ועל הקבצים שהוא מכיל:

e2e

תיקייה זו מיועדת לבדיקות מcka (end to end)

node_modules

תיקייה זו תכיל את כל הספריות שנוריד מ מקום. כל ספרייה נוספת שנתקין תישמר בה.

src

זהו התיקייה בה אנחנו נכתוב את קוד האפליקציה, פירוט נוסף בהמשך.

.editorconfig

קובץ המיועד להגדרות העבודה על ה IDE, סביבת העבודה.

.gitignore

מיועד למי שעבוד מול git, מוגדר בו ממה להתעלם כאשר מעלים את הקוד לgit.

angular.json

זהו קובץ חשוב מאד, הוא מנהל בעצם את כל האפליקציה דרך זה. בקובץ זה מוגדר היכן יפתחו פרויקטים חדשים בתחום ה solution (newProjectRoot) (solution).

כל פרויקט קיים (projects) מוגדר שמו וסוגו (האם אפליקציה או library) מהי תיוקית השורש שלו (sourceRoot) ולאן הקבצים לאחר build יעברו (outputPath).

בנוסף לזה מוגדר מהו הקובץ index אשר יש להציג בדף单一 page (single page), וכן איזה קובץ הינו הקובץ הראשי (main) המתניע בתוכו את האפליקציה.

ישן הגדרות לקבצי CSS נלוויים ועוד הגדרות רבות שב"כ לא יהיה צריך לשנות אותן. במידה וה solution יכול כמה פרויקטים, יוגדר מהו הפרויקט בירירת המודול (defaultProject).

package.json

קובץ זה מנהל את שם הפרויקט וגירסתו (במידה ומוצאים אותו בספריה) והחלק החשוב ביותר שבו זה ניהול כל הספריות בהן הפרויקט תלוי.

devDependencies - dependencies

ניתן להבחין בין מקטע dependencies למקטע devDependencies בו יוגדרו כל הספריות בהן הפרויקט עושה שימוש וחויבת עליהן להימצא יחד עם קוד האפליקציה כיוון שהקוד עושה בהן שימוש. כמו למשל ספריות האנגלולר או zx.js. לעומת זאת, devDependencies מגדיר את הספריות אשר בזמן הפיתוח יש בהן צורך, כמו למשל typescript ו cli, ספריות אלו נועדו לפעולות בזמן פיתוח, אולם לאחר שהקוד מוכן לשימוש אין בהן צורך.

scripts

דבר נוסף שניתן לראות בקובץ זה, זה ה scripts. חלק זה מיועד עבור כתיבת סקורייפטים מותאמים אישית, והרצתם ע"י פקודות חישום קומחה עם שם הסקורייפט שהגדכנו. יכולת זו מקלת על השימוש בסרט פקודות רצופות וארוכות, שנitin לבצען ע"י הרצת שם הסקורייפט בלבד.

ספרי גירסאות

ניתן להבחין כי בהגדרת הספריות - לכל ספריה מוגדרת הגירסה שלה, לעיתים כמספר קבוע, לעיתים עם תחילית ^, ולעתים עם תחילית ~.

כאשר נרצה גירסה קבועה של ספריה – נכתב את מספר הגירסה בלי אף תחילית. כאשר נרצה גירסה מג'ורית מסוימת של הספריה, אבל את העדכון האחרון שיצא בה – נכתב את מספר הגירסה עם התחילית ^ שאומר שגירסת הספריה שתרדת תהיה גדולה ככל שתימצא לאחר הספרה הראשונה של מספר הגירסה.

כאשר נרצה את הגירסה الأخيرة ולא משנה אם היא גירסה מג'ורית אחרת נכתב את מספר הגירסה עם התחילית ~ וכך נקבל תמיד את הגירסה الأخيرة שיצאה.

כל זהגורם לנו למצב בו יש לנו הגדרות לספריות בהן אנו תלויות ואולם הספריות בפועל לא בהכרח שתהיינה זהות להגדרות אלו (בגלל התחליות אם קיימות) כדי לאמת את מצב הספריות בזמן נתון, נוצר הקובץ:

package-lock.json

בקובץ זה מתוארכות המדיוקיות בהן הפרויקט עושה שימוש בספריות לאחר ה התקינה של הספריות לפי המוגדר ב package.json.

tsconfig.json

קובץ זה משמש את הקומפיילר של typescript ומנהה אותו כיצד לבצע את הקומפיילציה ל javascript. למשל לאייזה תקן להמיר את הקוד – es6 או es5.

src

לאחר שעברנו על המבנה הראשוני, ניתן להיכנס לתיקיית `src`.

בתיקייה זו תהיה תיקייה ה `app` – בה ימצא כל הקוד אותו נכתב.

ניתן כבר לראות עצת כי יצירת הפרויקט הכניסה בבסיסה מודול ראשי (`app.module.ts`) וקומפוננטה ראשית (`app.component.ts`) שהקוד שלהם נמצא תחת תיקייה זו.

***פירוט נוסף** בנושא מודול ראשי וקומפוננטות.

נראה גם את תיקיית `assets` זויה תיקייה לכל הישיות הנלוות לקוד, כמו למשל קבצי שפות, תמונות ועוד.

תיקיית `environments` מנהלת את מפתחות הערבים המשתנים לפי סביבות. יהיה בה קובץ "יעודי" לכל סביבה (פיתוח בדיקות ויצור)

שני קבצים נוספים וחשובים אלו ה `index.html` והקובץ `main.ts`. קובץ ה `index.html` זהו הקובץ היחיד בו האפליקציה תחל את פעילותה בדף, בה מיטען הקומפוננטה הראשית שהוגדרה וממנו והלאה הכל יתגלה בתוכו. (`spa` – single page application) קובץ `ts` יפורט בהמשך בנושא המודול הראשי.

הרכבת הפרויקט

בשביל להריץ את הפרויקט יש להקליד את הפקודה `ng serve`

פקודה זו תרים שרת מקומי `localhost:4200` ותריץ את הפרויקט עד לתוצאה המושלמת בדף.

כל עוד הפקודה חייה, או יקמפל את הקוד חדש עבור כל שינוי שתבוצע בקובץ כלשהו, הדף יתרען ויעבד לפי הקוד החדש שהגיע אליו. כל זה יקרה באופן אוטומטי.

הדף אותו אנגולר יריץ בדף `index.html` יהיה דף זה הוגדר לו כדף `index` בקובץ `angular.json`

ה או יזריק לתוכו מספר קבצי `script`, ביניהם יהיה הקובץ `js/main` המכיל את כל הקוד של האפליקציה (פירוט בהמשך)

Angular cli ויצירת רכיבים חדשים

כאשר נרצה להוסיף מודול חדש או קומפוננטה חדשה, תוכל לעשות זאת בצורה ידנית ע"י הוספת קובץ `ts` בשם הרצוי וככיתבת הקוד כלו.

ישנה חסيبة מיוחדת לקונבנצייה של שמות הקבצים. לכל קובץ יש לתת בשם תבנית המשקפת אותו, וזה יהיה כך: שם הקובץ + מהות הקובץ + סימנת הקובץ. שם הקובץ יהיה תמיד באותיות קטנות.

לדוגמה: מודול ראשי `appModule` יהיה בקובץ בשם צהה: `ts/app.module.ts`

קומפוננטה ראשית `appComponent` תהיה בקובץ בשם צהה: `ts/app.component.ts`

Angular v.7

קובץ html שיישמש את הקומפוננטה הזו ייקרא: app.component.html.ts

ניתן להיעזר ב cli angular לצורך כך, וליצור בקלות הרבה קומפוננטות/מודולים/שירותים וכל סוג נוסף של מחלקות/פרויקטים, נקלט בעזרת כל זה גם את הקוד הבסיסי מבלי שנצחטרך לכתוב זאת שוב ושוב, נקבל גם השלכות אוטומטיות על הפרויקט כתוצאה לכך, כמו למשל – ביצירת קומפוננטה דריך זו היא תתווסף באופן אוטומטי למערך declarations של המודול תחתו היא ממוקמת וכו'.

לזה cli ישנו סט של פקודות המתעדכנות מגירסה לגירסה ומוסיפות יתרון גדול לעבודה איתן.

ניתן לקרוא על הנושא בהרחבה באתר: <https://cli.angular.io/>

מודולים / מודול ראשי - Root Module

כל מודול יכתב כמחלקה שתזכה עם `@NgModule` זו מסמנת מחלקה כמודול.
כל מודול כולל בתוכו את כל המרכיבים: מרכיבי הציגות, השירותים, תתי מודולים ועוד'.

המאפיינים העיקריים בכל מודול:

declarations

הצהרות על שימוש במחלקות תצוגה שנוצר במודול זה (קומponentות או directives)

imports

"יבוא מחלקות (של מודולים בלבד) הנדרכים עבור מודול זה.

providers

הגדרת ספק השירותים שנוצר עבור המודול.

bootstrap

מאפיין זה נמצא רק במודול הבסיס הראשי של האפליקציה, הגדרה זו בעצם מתניעה את כל פעילות האפליקציה. מכאן ואילך לאו כבר יכולות פנימיות של המודול. ההתנווה היא אחת עבור כל האפליקציה והיא מתניעה את המודול הראשי, פירוט נספּ בהמשך – מודול ראשי.

exports

מלבד הגדרת המודול, אם נרצה שמודול זה יוצג יוכל לשמש במודולים אחרים נדרש להגדיר מה מתוך המודול מיועד ליצוא כרךשמי שיצור את המודול קיבל גם את מה שנדרש במערך זה מבלי לכתוב זאת במפורש.

AppModule - RootModule

מודול השורש תפקידו לומר לאנגולר איך לבנות ולהתניע את האפליקציה. מודול זה יקרא לרוח `AppModule`.

בכל אפליקציה יהיה לפחות מודול אחד – מודול הבסיס. ככל שהמערכת תהיה גדולה יותר כרך יהיה בה יותר מודולים, אולי `AppModule` יהיה תמיד, והוא גם יהיה המודול הראשי שתתניע את האפליקציה.

להלן הקוד המינימלי ביותר שיש לכתוב עבור `AppModule`

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
```

Angular v.7

```

imports: [ BrowserModule ],
declarations: [ AppComponent ],
bootstrap: [ AppComponent ]
})
export class AppModule { }

```

import

פקודה זו גורמת ליבא קובץ. יבוא זה גם מאפשר לנו השלמה אוטומטית של Typescript לצורך נוחות והכרת האובייקטים אותם אנחנו עושים.

יבאנו את `NgModule` בכך שונכל להשתמש בdecorator שנקרא `@NgModule` עבור הגדרת מחלקה כמודול.

יבאנו את `BrowserModule` בכך שונכל להריץ את האפליקציה בדף.

יבאנו גם את `AppComponent` שזו תהיה המחלקה של הקומפוננטה הבסיסית אותה נכתב בעצמו ונתני ע"י המודול.

@NgModule

הצהרה זו היא מסוג `decorate`, דומה ל `attribute` – אנחנו בעצם "מקשטים" את המחלקה בתוספות שונות. במקרה הזה אנו מוסיף על הגדרת המחלקה שהיא תשמש גם כמודול. בשביל שהיא תנתנו כמודול ישנו metadata שעליינו לספק לאנגולר:

imports

כל מודול שנכתב, או כל מודול שקיים כבר – אם נרצה להשתמש בו בקוד נהיה חיבים ליבא אותו באמצעות `imports`.

חשוב!! רק מחלקות שנכתבו כ `@NgModule` ייכנסו לטור מערך `imports` זה. אין לשים שם אף סוג מחלקה אחר.

הבדל בין `import` שאנו כתבים בתחילת הדף – import הוא פקודה של javascript בלבד, והיא מייבאת קבצים שונים ונותנת הכרה לנוטונים בהםום כך שתאפשר גם השלמה אוטומטית. אין קשר בין `Import` לאנגולר. לעומת זאת `imports` זהו imports של `NgModule` והוא אומר לאנגולר שעליינו להשתמש ביכולות הקיימות במודולים אלו.

שנעם הרבה מודולים קיימים באנגולר כמו למשל `BrowserModule` או `HttpModule` וגם `RouterModule`.

המודול `BrowserModule` למשל הוא מודול שקיים בכל אפליקציה שרצה על דף.

declarations

חיבים להציג על כל קומפוננטה במודול אחד (ורק אחד). יש לכתוב את רשימת הקומפוננטות המשתתפות במודול זה בטור של `declarations`. כל קומפוננטה שנוסף במהלך האפליקציה – נהיה חיבים להציג עליה בטור זה.

חשוב!! רק קומפוננטות, directives או pipes ייכנסו לטור מערך `declarations`. אין לשים שם אף סוג אחר של מחלוקת (לא `Model`, `Module` ולא `Service`)

bootstrap

כאן אנו בעצם אומרים לאנגולר מה אנחנו רצים להתנייע במודול זהה. תהליך ההתניעה ייצור את הקומפוננטה/וות שהוגדרו במודול זה ייכניס אותו ל DOM.

Bootstrap זה אינו קשור כלל לספריית bootstrap של עיצוב.css.

מקובל ליצור קומפוננטה ראשית שאותה נתנייע במודול הראשי, וממנה ולהלאה כבר נגיע לקומפוננטות נוספות. לקומפוננטה ראשית מקובל לקרוא AppComponent.

במודול שאינו ראשי יש את נושא exports, entryComponents ועוד.

איך מתנייעים את המודול עצמו?

ישן כמה דרכים, המומלצת היא דרך יצירת קובץ main.ts בצורה זו:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

קוד זה מקמל ומטנייע את המודול הראשי שלנו, מייצר מופיע של מחלקת הקומפוננטה ומכניס את הselector template לתוך התג אותו הגדרנו כ selector של הקומפוננטה שנמצא בדף html,index.html, ומציג אותו על כל תכונותיו.

מכאן ולהלאה, תוכן הקומפוננטה מנוהל בידי אנגולר, לפי התנהגות הקומפוננטה.

קובץ זה main.ts מיוצר פעמי אחד, ואין צורך לעורך לשנות אותו.

Lazy Loading

טעינה עצילה של מודולים/קומפוננטות – כאשר אין צורך לטעון מראש קוד שלא בהכרח יבוא לידי שימוש, ניתן להימנע מטעינה שלו, ולטעון אותו רק כאשר יהיה צורך.

פירוט נרחב על הנושא נמצא בהמשך, מומלץ לקרוא לאחר הבנה של פרק ה routing.

Components - קומפוננטות

כאשר ניגשים לכתוב רכיבי תצוגה, נשתמש בקומפוננטות.

קומפוננטה זהה בעצם חילה אחת המרכזת בתוכה את כל מה שהרכיב צריך.

קומפוננטות מכילות בתוכן שני חלקים – חלק לוגי וחלק ויזואלי.

החלק הלוגי – זהו בעצם המחלקה עצמה שתכיל את כל המאפיינים והמתודות.

החלק הוויזואלי – יהיה תבנית html שתוצג בדף, תבנית זו תעשה שימוש בתנומי המחלקה.

כאשר אנחנו כתבים מחלוקת, אנו בעצם מייצרים קוד JS טהור, אשר בinternals אין בו קשר לאנגולר, אם נרצה שאנגולר יבין שזו קומפוננטה ויקשר אותה למחלוקת זו נצטרך להשתמש ב metadata.

ליצור הגדרת קומפוננטה נוסיף מסוג `@Component` decorator וניתן את כל המידע שאנגולר צריך לצורך לייצר מזה קומפוננטה.

העיקריים בקומפוננטה: **metadata**

selector

צורת המופע שיוצר לקומפוננטה זו. יוצרת-tag בשם `selector` שנitin, ונוכל על ידי כך לשתול בכל html אלמנט עם סלקטור זה ולקבל על ידי כך את כל מה שהקומפוננטה תשופר בסופו של דבר.

```
@Component({
  selector: 'app-root',
  ...
})
export class AppComponent { }
```

template

בכל metadata של קומפוננטה קיים התוכן שלה, ע"י ה `.template`.

הtemplate מקבל מחuzeות של תצוגה – מה שאומר html כמחuzeות שיצג על הדף.

כאשר המחזזה של ה template ארוכה ואנו רוצים לרדת שורה, ניתן לבצע "חיבור" בין השורות למחזזה אחת ארוכה ע"י שימוש בסימן ` (back tick) בתחילתה ובסוף המחזזה.

```
@Component({
  selector: 'app-root',
  template: `<h1>hello app component</h1>
    <p>text and more</p>
    <div>more text and elements</div>
    <button>send</button>`,
  ...
})
```

Angular v.7

```
export class AppComponent {  
}
```

templateUrl

כתובת יעד של מסמך התצוגה.

כאשר נרצה ליצור תוכן ארוך ומורכב, לא יהיה נכון לכתוב אותו כמחרוזת, מלבד זאת, יכולות להיות לנו טוויות בכתיבה יהיה לנו קשה להעתן כאשר מדובר בהרבה תוכן שנראה כמו באותו צבע ותונועה. לצורך כך, יוכל לכתוב את התוכן בדף html לכל דבר, ולתת את הכתובת שלו בvariable templateUrl. שקיים ב metadata של הקומפוננטה, במקום לתת את המחרוזת ל templateUrl.

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  ...  
})  
export class AppComponent {  
}
```

styles

ניתן להגיד את ה css במפורש בתוך המחלקה עצמה. כך:

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styles:[`h1{ color: red; } p{ text-align: right; }`]  
})  
export class AppComponent {  
}
```

לחלופין ניתן להגיד מערך של קבצי css בהם ניתן יהיה להשתמש על ידי:

styleUrls

מערך של קבצי css שהקומפוננטה תעשה בהם שימוש.

לא ניתן להשתמש בשתי ההגדרות, או styles או styleUrls.

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.scss'],  
}
```

```
})
export class AppComponent {
}
```

providers

שירותים שקומפוננטה זו תצורך נצטרך להגדיר ע"י providers בשביל שאנגולר יזקן לנו אותם במבנה של המחלקה ונוכל להשתמש בהם.

encapsulation

על כל קומפוננטה ניתן להגדיר האם ה css שמודדרים עליה ישפיעו על יתר הרכיבים או לא, והאם היא עצמה תושפע מרכיבים אחרים.

ישנם 3 סוגי:

encapsulation: ViewEncapsulation.Emulated

encapsulation: ViewEncapsulation.Native

encapsulation: ViewEncapsulation.None

– יורש css אך אינו מוריש וזהו ברירת המחדל. Emulated

– מתיחס אך ורק ל css שמודדר עליו, אינו יורש ואין לו מוריש. Native

– הנקודות כמו שהיינו מצפים ב css, שם יש שינוי, השינוי ישפיע על כל הדף. None

Data binding

משמעות המושג הינה הקשר בין החלק הלוגי לחלק היזואלי – קישור בין נתונים המחלקה ל html שלה.

ישנם 4 סוגים עיקריים של קשר בין הлогיקה/המידע לבין ה html.

String binding

כאשר נרצה לחבר בצורה הבסיסית ביותר בין נתונים כלשהו מהמחלקה לבין התצוגה שלו ב html תוכל להשתמש בצדד סוגרים מסוימות ולכתוב בתוכן את המאפיין אותו אנו רוצים להציג.

לדוגמה:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
  `
})
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

בדוגמה זו יש לנו מחלוקת בשם AppComponent שמכילה בתוכה שני מאפיינים: title ו-myHero. בtemplate של הקומפוננטה רשemoות תגיות כותרת h1 ובתוכה שמו צמד סוגרים מסוימות ורשemoות את שם המאפיין title – מה שזה יעשה, בזמן ריצה, אנגולר יקח את ערך המאפיין title וישים אותו במקום כל הפעם. זהו סוג של bind. כאשר ערך זה ישנה – התצוגה גם היא תשתנה מבלתי שנדאג לכך. באוטה צורה בדוגמה זו גם רשemoות את ערך המאפיין myHero בתוך המשפט כותרת h2.

סוגרים מסוימות מכילות string בלבד.

ניתן להכניס בתוכו משתנה המחזיר string או משתנה המחזיר מספר שיכל להיות מוצג ע"י המירה ל string. ההמרה תבוצע אוטומטית.

ניתן אף להכניס string רגיל ופשוט {{'hi'}}.

ניתן אף לקרוא לפונקציה, שתחזיר string או ערך אחר (מספר למשל) שיכל ע"י המירה להיות string. לדוגמה {{()}} כאשר הפונקציה מחזירה getName().string

ערך של משתנה בוליאני מקבל המירה אוטומטית למיללים true/false.

לא ניתן להכניס פקודות או התניה כגון for, if וכו'.

Property binding

שינויי ערכי ה properties בהתאם לנתחים הנמצאים במחלקה. כתוב זאת ע"י סוגרים מרובעתות.

לדוגמה, כפטור יהיה disable בהתאם לערך `isRight`:

```
<button [disabled]="isRight">press</button>
```

Event binding

שימוש בקוד הנמצא במחלקה כתגובה לאירוע על אלמנט. פירוט נרחב בהמשך בנושא האירועים.

Two way binding

כאשר יש צורך לתקשר בין אלמנט למשתנה ובין המשתנה לאלמנט.

לדוגמה, בתגית `input` יש צורך שבעת הקלדה ו שינוי הטקסט של ה `input`, יקבל המשתנה אליו הוא מוקשור - את הערך המעודכן. כמו כן להיפר, בשינוי הערך של המשתנה ישנה השניה של האלמנט `Input`.

הקשר נכתב כר `[(ngModel)]="nameOfProperty"`
חומר זה יורח בהמשך בנושא הטעסים.

Constructor

בדוגמה לעיל יוצרים מאפיינים למחלקה `AppComponent` ואיתחולו אותם במשפט ההצהרה עליהם. באוטה מידヤ יכולנו רק להציג עליהם, ולא תחולו אותם בכל זמן אחר. אם נרצה למשל לאותם במבנה של המחלקה, יוכל לעשות זאת ע"י שימוש בפונקציה המובנית `constructor`.

לדוגמה:

```
export class AppComponent {
  title: string;
  myHero: string;
  constructor() {
    this.title = 'Tour of Heroes';
    this.myHero = 'Windstorm';
  }
}
```

*ngFor - הצגת מערכיים

כאשר נרצה להציג מידע מסוים – נבצע בעצמו איזושהי פעולה שתעביר על המערך ותציג את האיבר הנוכחי. אנגולר מאפשר לנו לבצע לוולה זו לצורה פשוטה ביותר ע"י שימוש ב directive בשם `repeat` בשמם `.ngFor` הוא בעצם `repeater`, שייחזור על האלמנט בו הוא נכתב, מספר איברי המערך.

Angular v.7

Take each hero in the heroes array, store it in the local hero variable, and make it available to the templated HTML for each iteration.

המחלקה תיראה כך:

```
export class AppComponent {
    title = 'Tour of Heroes';
    heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
    myHero = this.heroes[0];
}
```

התצוגה תיראה כך:

```
template: `
<h1>{{title}}</h1>
<h2>My favorite hero is: {{myHero}}</h2>
<p>Heroes:</p>
<ul>
    <li *ngFor="let hero of heroes">
        {{ hero }}
    </li>
</ul>
`
```

חשוב לציין כי השם `hero` שופיע בדוגמה מופיע בקובץ `heroes.ts`, כלומר מושג זה מוגדר בפונקציית `getHeroes()`.

`ngFor` בדוגמה זו הציג מערך, אולם הוא יכול לשמש עבור כל אובייקט איטרציוני.

יצירת מחלקות נתונות

עד כה יצרנו מאפיינים פשוטים בקומפוננטות, כאשר אנו נדרשים למאפיינים מורכבים יותר עלינו ליצור מחלקות. יצירת מחלקות הינה חלק משפט typescript אין לה קשור ישיר לאנגולר. בקומפוננטה אנו נגדיר מאפיין מסווג המחלקה שנוצר וכך הקוד יראה נקי ויעיל יותר.

את המחלקה נוצר כקובץ `ts`, נזהיר על מחלקה ונגדיר אותה כ `export` בשביל שאנגולר יוכל לקבל אותה.

לדוגמה:

```
export class Hero {
    public id: number;
    public name: string;
}
```

יצורנו מחלקה בשם `hero`, הגדרנו לה שני מאפיינים. המאפיינים צריכים להגדיר בנוסף לשם את סוגם והאם את פרטיהם/ציבוריים.

ניתן ליצור זאת בקיצור ע"י שימוש בבניאי כך:

Angular v.7

```
export class Hero {
    constructor(
        public id: number,
        public name: string) { }

    }
    צורת כתיבה זו תצהיר על בניית המקביל פרמטרים אלו ואת סוגם, תצהיר על משתנים ציבוריים
    בשמות זהים ותאפשר לטור אותם משתנים את הפרמטרים שקיבלה בעט יצירה מופיע מחלקה זו.

    לאחר שייצרנו מחלקה זו נוכל להשתמש בה בקומפוננטה ולהחיליף את המערך שהגדכנו במערך מסווג
    המחלקה hero:
```

```
heroes = [
    new Hero(1, 'Windstorm'),
    new Hero(13, 'Bombasto'),
    new Hero(15, 'Magneta'),
    new Hero(20, 'Tornado')
];
}
```

*ngIf - תנאי להציג

אילוג הוא directive של אングולר, המאפשר לנו לבצע הוספה/ הסרה של אלמנט בהתאם לקיום תנאי כלשהו.

לדוגמא:

```
<p *ngIf="heroes.length > 3">There are many heroes!</p>
```

התנאי יכתב בתוך מרכאות.

התנאי יכול להיות כל ביטוי שהוא, על מאפיין אם קיימ, אם שווה לערך כלשהו ועוד. חשוב לציין שאנגולר לא יציג או יסתיר את האלמנט אלא יוסיף אותו או יסיר אותו מה DOM ממש.

לסיכום, שלושת השלבים ליצירת קומפוננטה:

1. יצירת מחלקת הקומפוננטה כולל כל המידע הנדרך לשם כך.

יש צורך להוציא תיקיה עבור קובץ המחלקה, ה html וה css במידה יהיה קיימים.

2. הוספת קומפוננטה זו לטור מערך declarations של המודול בו היא נמצאת.

3. שימוש בפועל בקומפוננטה: כתיבת תגיית עם סלקטור כפי שהוגדר לקומפוננטה זו - במקומות אותו נרצה לראותה.

אירועים

בדומה לארועי DOM אングולר מאפשר גם הוא לבצע bind לארועים, כאשר הbind מתיחס למетодה הקיימת בקומפוננטה. לדוגמה:

```
<button (click)="onClickMe()">Click me!</button>
```

בדוגמה זו ניתן לראות כי ע"י () אנו מגדירים bind מהאלמנט לקומפוננטה. שע"י אירוע לחיצה תיקרא המethodה onClickMe של הקומפוננטה.

אירועים יכולים להיות אירועי מקשיים, פעילות עכבר, לחיצות כפתורים וגעית מסך.

כל האירועים שקייםם ב5 html קיימים גם באנגולר ונitin להשתמש בהם. (מורידים את החס-ומכניותם לתוך סוגרים עגולות)

\$event object

אובייקט \$event מביא לנו נתונים מהאירוע הנוכחי, שיכולים לתת לנו מידע ולהשפייע על הקוד שלנו. בכל אירוע, אובייקט זה יביא בהתאם את הנתונים הרלוונטיים, בדומה לאובייקט event שאמנו שולחים בכתיבה html רגיל אשר מוכר בDOM.

דוגמה לשימוש באובייקט זה:

:template

template:

```
<input (keyup)="onKey($event)">
<p>{{values}}</p>
```

בקומפוננטה:

```
export class KeyUpComponent_v1 {
  values = '';
  onKey(event:any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

ניתן לשים לב שהגדכנו את הפרמטר של event ללא type, עשינו זאת ע"י הגדרתו כany.

במקרה ואנו לא יודעים מראש את סוג האובייקט שיגיע אלין, אנו יכולים להשתמש ב Type זה ולקלוט לתוכו את כל טיפוסי הנתונים.

דוגמה זו עשינו שימוש ב event.target הנמצא באירועים על שדות קלט ומתייחס בד"כ לאלמנט עצמו.

בדוגמה זו ניתן לשים את key של המKeySpec עליון המשמש לחץ וכן הלאה.

Angular v.7

דוגמה לימוש אירוע עם event מסוג KeyboardEvent:

```
export class KeyUpComponent_v1 {
  values = '';
  onKey(event: KeyboardEvent) { // with type info
    this.values += (<HTMLInputElement>event.target).value + ' | ';
  }
}
```

בדוגמה זו נתנו type לאובייקט, ולקן נדרש להמיר את event.target ל HTMLInputElement מכיוון שלא לכל ה-target יש value, מכיוון שהוא מתעסקים כאן כבר עם טיפוס נתונים, אנו לא יכולים לבצע כל קוד, אלא רק קוד שיעבור קומפונלציה, ולכן במקורה ולא כל target יכול value אנו חייבים להמיר לטיפוס הנתונים אליו אנו מתכוונים ועלינו לבצע את הקוד.

template reference variable

כאשר אנו רוצים לבצע פעולה על אלמנט לקומפוננטה, אנו עושים זאת ע"י bind של אירוע. אולם אם נרצה לבצע פעולה על אלמנט בתוך ה-template עצמו או לקבל על אלמנט נתונים לאו דזוקא ע"י bind לAIROU, נוכל לעשות זאת ע"י reference variable – הגדרת משתנים בתוך ה-template.

בדוגמה הבאה נראה כיצד הגדרנו משתנה בשם box על האלמנט של שדה הקלט, ובהמשך ה-template ניגשנו ל-value.of_box כדי להציג אותו, box שהוגדר על האלמנט קיבל אליו הצבעה לאלמנט וכאש ניגשנו ל-value.of_box בפועל בעצם את הערך של שדה הקלט:

```
template: `
<input #box (keyup)="0">
<p>{{box.value}}</p>
`
```

חשוב לציין שהקומפוננטה אינה מכירה משתנים אלו, אלו משתנים שמוגדרים ב-template ורק שם הם מוכרים ונגישים.

נקודה נוספת, כל זה יעבוד רק במידה והגדרנו bind ל-AIROU מסוים, מכיוון שלא הגדרת bind אנגולר לא יפעיל את עצמו ויזהה את השינויים המבצעים.

במקרה זה הגדרנו bind מינימלי "0"=(keyup) שזוהי הצורה הקצרה ביותר.

ניתן לשלוות את ערכי המשתנים הללו לקומפוננטה וכך לקבל קוד נקי בקומפוננטה שאינו צריך לעבוד על האובייקט event, לדוגמה:

```
@Component({
  selector: 'key-up2',
  template: `
<input #box (keyup)="onKey(box.value)">
<p>{{values}}</p>
`
```

Angular v.7

```
}
export class KeyUpComponent_v2 {
  values = "";
  onKey(value: string) {
    this.values += value + ' | ';
  }
}
```

key event filtering

כאשר אנו רוצים לבדוק האם הוקש מקש האנטר, לרוב אנו מבצעים זאת ע"י בדיקת ה key שהוקש, דבר זה מחייב Maiyanנו לכתוב קוד שיקרא עבור כל הקש מקש כלשהו ובדיקה ה key שלו, במקומ להגדיר אירוע שיקרא אך ורק בלחיצת אנטר, אנגולר מאפשר זאת ע"י pseudo-event שנקרא keyup.enter

לדוגמה:

```
@Component({
  selector: 'key-up3',
  template: `
    <input #box (keyup.enter)="onEnter(box.value)">
    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v3 {
  value = "";
  onEnter(value: string) { this.value = value; }
}
```

נקודות חשובות לציין:

- יש להגדיר template variables על אלמנטים (ניתן להציב להם ערכים שונים, אולם ההגדרה תבוצע על אלמנט)
- כאשר מעבירים משתנים אלו – יש להעביר את ערכיהם ולא אותם עצם.

Input

כאשר יש לנו צורך לקבל את ערכו של מאפיין מסוים בקומפוננטה מקור חיצוני יוכל לבצע זאת ע"י הגדרתו כ `task` – כמשתנה המיועד לקלוט נתונים.

הצורה בה נגידיר זאת תהיה ע"י `@input():decorate()` (בדומה להגדרת attribute על משתנה)

בנוסף לכך, מי שיצור קומפוננטה זו וירצה לבצע השמה למשתנה זה יבצע זאת ע"י `bind` רגיל למשתנה בשם זה.

לדוגמא, אם ניצור קומפוננטה המציגת פרטי משימה, מחלקת הקומפוננטה תיראה כך:

```
@Component({
  selector: "task-details",
  templateUrl: "task-details.component.html",
})
export class TaskDetails {
  @Input()
  task: Task;
}
```

כאשר נשתמש בקומפוננטה זו נשלח אליה את המשימה אותה אנחנו רוצים שהיא תציג בצורה צזו:

```
<task-details [task]="selectedTask"></task-details>
```

יש לשים לב שהbind בוצע עבור המשתנה בשם `task` שהוגדר כ `task` במחלקה.

במידה ונבצע bind למשתנה אשר לא הוגדר כ `task`, תהיה לנו שגיאת ריצה.

Output

אחד הדברים המזוהים באנגולר, הוא האפשרות לייצר אירועים. בדומה לאירועים מובנים שקיימים כמו אירועי לחיצות או אירועי מקלדת וכד', כך יש לנו אפשרות לייצר עצמן אירועים אירועים.

אירועים הם בעצם פלט כלשהו מן הקומפוננטה אל החוץ, מי שנרשם לאירוע מקבל אליו נתונים כלשהם הקשורים לאירוע והוא בוחר מה לעשות איתם. הקומפוננטה פולטת החוצה נתונים דרך הרשמה לאירועים, כאשר אירועים אלו מתרחשים – מי שנרשם אליהם מקבל את הנתונים כ `event` `.object`.

לדוגמא, אם ניצור קומפוננטה המציגת פרטี้ מסוימות וקומפוננטה המשימה פרטוי משימה. קומפוננטת הרשימה תשלח לקומפוננטת פרטוי המשימה את המשימה הרלוונטיות וקומפוננטת פרטוי המשימה תציג את פרטיה. כפתר השמירה יימצא בתוך קומפוננטת פרטוי המשימה, רק כאשר ילחוץ עליו נרצה לבצע את השמירה לרשימה. כאן יש לנו בעיה, כיוון שהכפתר נמצא בקומפוננטת פרטוי המשימה, ואילו הרשימה עצמה לא נמצאת בקומפוננטה זו כדי שנוכל לעמוד אותה כתוצאה מן הלחיצה על כפתר זה.

air פותרים בעיה זו? יוצרים `output`.

הreasון הוא פשוט: כאשר כפתר השמירה ילחוץ, קומפוננטת פרטוי המשימה תרים אירוע `shikra` `onSaveTask` אירוע `onSaveTask` זה יפלוט החוצה את נתונים המשימה המיועדת לשמירה. מי שיירשם לאירוע – כאשר אירוע זה יקרה יקבל אליו את נתונים המשימה והוא יפנה לקוד אצלו שיעשה עם זה מה שהוא רוצה. במקרה שלנו – קומפוננטת רשות המשימות תירשם לאירוע `onSaveTask` והאירוע זה יקרה היא תפנה לפונקציה שנמצאת אצלה ובה תבצע את העדכן לרשימה.

air עושים את זה?

קודם כל יש להגדיר משתנה לאירוע מסווג `EventEmitter` נגדיר אותו כך:

```
onSaveTask: EventEmitter<T> = new EventEmitter<T>();
```

הגדרנו בעצם משתנה בשם `onSaveTask` מסווג `EventEmitter` על type כלשהו.

למה חייבים את ה `type`? יכול להיות כל סוג, גם `any`. `type` זה מהוות את החתימה של האירוע. כלומר, אירוע זה פולט נתונים מסוימים, מי שיירשם אליו ידע שהוא קיבל תמיד מן האירוע הזה נתונים מסווג זה שהוגדר.

הגדרה נוספת שעליינו לבצע על משתנה זה – זהdecorate של `output`. לסמן שהוא בעצם מיועד לפולט חיצוני. נעשה זאת כך:

```
@Output()  
onSaveTask: EventEmitter<T> = new EventEmitter<T>();
```

עתככ כל מה שנותר הוא להחליט מתי האירוע קורה, להפעיל אותו ולשלוח את הנתונים הרצויים.

בדוגמה שלעיל, כתוצאה מלחיצה על כפתר שמירה נפנה לפונקציה בקומפוננטה, הקוד בפונקציה ירים את האירוע. הרמת האירוע מתבצעת ע"י פקודת `emit` למשתנה `theEventEmitter`.

הכפתר בקומפוננטת פרטוי המשימה:

Angular v.7

```
<div class="btn btn-success" (click)="saveTask()">
  <span class="glyphicon glyphicon-floppy-save"></span> Save
</div>
```

קומפוננטת פרטיה המשימה:

```
@Component({
  selector: "task-details",
  templateUrl: "task-details.component.html",
})
export class TaskDetails {
  @Input()
  task: Task;

  @Output()
  onSaveTask: EventEmitter<Task> = new EventEmitter<Task>();

  saveTask() {
    this.onSaveTask.emit(this.task);
  }
}
```

קומפוננטת רשיימת המשימות צריכה להירשם לאירוע בצד קבל את כל הפלט זהה. הרישום לאירוע מתבצע בכל אירוע רגיל אחר: סוגרים עגולות עם שם האירוע = שם הפונקציה בקומפוננטה.

(onSaveTask)="saveTaskToList(\$event)"

בצד קבל את ה event object שהאירוע הcin עברונו, יש להשתמש במילה השמורה event (בדומה לאירוע DOM) רק בתוספת ה \$.

לדוגמא:

```
<task-details [task]="selectedTask"
  (onSaveTask)="saveTaskToList($event)"></task-details>
```

טפסים

אנגולר נותן לנו כלים מתקדמים לעבודה עם טפסים המקלים علينا בהרבה את העבודה.
ישן כמה צורות ושיטות עבודה לטפסים, נתמקד כרגע בשיטה הפשטota ביותר.

בשביל להתחיל לעבוד עם טפסים ולקבל יכולות אלו, علينا קודם כל לייבא את המודול FormsModule מתוך "@angular/forms" למודול הראשי שלנו.

```
import { NgModule } from "@angular/core"
import { BrowserModule } from "@angular/platform-browser"
import { FormsModule } from "@angular/forms"

import { AppComponent } from "../components/app.component"
import { TasksComponent } from "../components/tasks/tasks.component"

@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [AppComponent, TasksComponent],
  bootstrap: [AppComponent]
})
export class AppModule {
```

[(ngModel)]

הכלי הראשון והשימושי ביותר הוא ה `.ngModel`.

bind זו בעצם bind דו צדדי הפולט וקולט בו זמינות המשתנה אליו הוא מיוחס.

bind זו צדדי זה בעצם המושג two way binding. כלומר המשתנה משתקף דרך שדה הקלט, וכל שינוי בשדה הקלט משתקף למשתנה.

ngModel ניתן להגיד רק על שדות קלט, מהסיבה הפשטota של ngModel אין משמעות ללא קלט, כל הרעיון הוא שהקלט מעדכן מיידי את המשתנה, ואם השימוש יהיה לצורך תצוגה בלבד ניתן להשתמש ב `{}` one way binding רגיל.

ngModel נגדיר כך:

```
<input type="text" id="name" name="name" [(ngModel)]="task.name" />
```

אם אנחנו עובדים עם טופס אמיתי, כלומר עם תגי `form`, חובה علينا להוסיף עבור כל שדה קלט את המאפיין `name`.

המאפיין `for` מגדיר תגי `label` המקשרת את ה `for` לפ'י `id`.

```
<label for="name">Task Name</label>
```

המאפיין `name` מגדיר עבור אינדוקס שדה הקלט לצרכים שונים.

מעקב אחר מצב השינויים והחוקיות עם ngModel

אנגלור נותן לנו הרבה מעבר ל way two binding כאשר אנו משתמשים עמו. ngModel אングולר מספק לנו מידע אודוט שדה הקלט – מה מצבו מבחינת השינויים שבוצעו בו ומבחןת הולידציה שלו. נתונים אלו מואחסנים ב ngModel המוגדר עבור המאפיין, ובאים לידי ביטוי גם בצורה מחלקות CSS מיוחדות שאנגולר מחייב על השדות. מצב השדה מתואר בטבלה מטה:

State	Class if true	Class if false
The control has been visited.	ng-touched	ng-untouched
The control's value has changed.	ng-dirty	ng-pristine
The control's value is valid.	ng-valid	ng-invalid

אם נגדיר `template reference variable` על שדה קלט כלשהו המיויחס ב `ngModel` למאפיין `model.name`, נוכל בקלות לצפות במחלקות אלו. נעשה זאת כך:

```
<input type="text" id="name" required [(ngModel)]="model.name"
       name="name" #spy>
<br>{{spy.className}}
```

הגדירנו משתנה בשם `spy` על ה `input`, הוא קיבל את תכונותיו הטבעיות של האלמנט. אחת התכונות של כל אלמנט זה `className`, מאפיין זה מכיל את רשימת הקלאסים הקיימים על האלמנט. בתחילת ראות שיש לנו `ng-untouched` שהוא אומר שלא נגע עדין בשדה, ברגע שנשים פוקוס על השדה, גם אם לא נשנה את ערכו – קלאס זה ישתנה ל `ng-touched` כלומר, נגע בשדה. את הקלאס `ng-pristine` נוכל לראות כל עוד לא נגע בערך המקורי של השדה, ברגע שנשנה שהוא בערך המקורי – קלאס זה ישתנה ל `ng-dirty` כלומר, למלכו את הערך המקורי וזה לא הערך המקורי, שאמו הוגדרה ולידציה כשליה, כאשר השדה יהיה ולידי נוכל לראות את הקלאס `ng-valid` ולהיפך – `ng-invalid`.

נוכל באמצעות קלאסים אלו לתת חיוי למשתמש על מצב השדה:

Dr IQ	Valid + Required
Chuck Overstreet	Valid + Optional
	Invalid (required optional)

© כל הזכויות שמורות לפניה אורן. אין לקרוא או להדפיס/לצלם או להעביר או להעתיק
בחומר זה בשום צורה ואופן ללא רשות כתובה ומפורשת. 052-7142050

```
.ng-valid[required], .ng-valid.required {
    border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form) {
    border-left: 5px solid #a94442; /* red */
}
```

הצגה/הסתירה של הודעות שגיאה בהתאם ל מצב ה חוקיות של המודול

אם ניקח את המשתנה שהגדכנו קודם, ונציב אליו את ה `ngModel` directive שהוא בעצם המכיל את כל המידע על המודול, יוכל לקבל את אותו מידע ויתר מכך.

דוגמא בסיסית להצגת הודעת שגיאה בתנאי שהמודול אינוolid לאחר שנגעו בו (כלומר אנו מסתירים את ההודעה אם המודולolid או שעדיין לא נגעו בו)

```
<label for="name">Name</label>
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel">
<div [hidden]="name.valid || name.pristine"
      class="alert alert-danger">
    Name is required
</div>
```

דיבוי ולידציות

במידה והגדכנו עבור פק'ד אחד כמה ולידציות, ונרצה בזמן אמיתי לדעת מה מצב כל ולידציה בפניהם, לא רק האם השדהolid או לא, אלא עבור סוג של ולידציה נרצה לדעת אם היא חוקית או לא. נשתמש בזה כאשר נרצה למשתמש להציג עבור כל ולידציה – הודעת הסבר שונה ומוקצת. נרצה להציג כל הודעה רק במידה והיא רלוונטית. נעשה זאת באמצעות שימוש errors באובייקט errors הנמצא לנו ב `ngModel`. נעשה זאת כך: (הסביר בהמשך)

```
<label for="name">Name</label>
<input type="text" id="name" required minlength="4" maxlength="24"
       name="name" [(ngModel)]="task.name"
       #name="ngModel" >
```

```
<div *ngIf="name.errors && (name.dirty || name.touched)">
  class="alert alert-danger">
    <div [hidden]="!name.errors.required">
      Name is required
    </div>
    <div [hidden]="!name.errors.minLength">
      Name must be at least 4 characters long.
    </div>
    <div [hidden]="!name.errors.maxLength">
      Name cannot be more than 24 characters long.
    </div>
</div>
```

יש לשימוש לב לכמה דברים:

- האלמנט `input` קיבל 3 ולידציות: `required`, `minlength`, `maxlength`.
- המשתנה שהגדרנו על האלמנט בשם `name` קיבל את `name.ngModel` המכיל את המידע לאנגלול על המודול.
- הגבלנו את הצגת בлок ההודעות בכללותן למקורה שקייםות שגיאות ע"י `if` על אובייקט `errors` של ה `name`.
- עבור כל הודעה ביצענו הצגה/הסתירה לפי קיומה של שגיאה מסווג הרלוונטי.

איתחול הטופס *reset form*

בשביל להחזיר את מצב הפקדים במצבו המקורי מבחינת הדגלים שסומנו לנו על מצב הקלט והחווקיות, נצטרך לאפס את הטופס ע"י הפקודה `reset`.

על התגית `form` נגדיר משתנה שיקבל אליו את ה `ngForm` – אובייקט הטופס המכיל את כל המידע אודות הטופס, כולל את היכולת לאפס אותו. נעשה זאת כך:

```
<form #taskForm="ngForm">
```

כאשר נרצה לבצע איתחול במצב הפקדים בטופס כתוצאה מלאיצה על הוספה רשותה או עריכת רשומה נעשה זאת באמצעות גישה למשתנה שהגדרנו:

```
<button type="button" class="btn btn-default" (click)="newTask();
taskForm.reset()">New Task</button>
```

ביצוע submit לטופס באמצעות ngSubmit

אנגולר מאפשר לנו להגדיר מה יקרה ב submit של הטופס בצורה כזו: (לא חובה, אבל ישנה אפשרות)

```
<form (ngSubmit)="onSubmit()" #taskForm="ngForm">
```

אירוע זה מפעיל את function onSubmit() שנקבע כפטור submit כלשהו בטופס.

הסימת שליחת הטופס כל עוד אין לו ידיים

אפשר לחסום את כפטור submit כל עוד הטופס לא לו ידיים בצורה פשוטה מכך:

```
<button type="submit" class="btn btn-success"
[disabled]="!taskForm.form.valid">Submit</button>
```

Services

שירותים, בכלל, הן בעצם מספקי יכולות כלשהן. הchl `reuse` של קוד וכליה בגישה לserver.

המושג service אינו בהכרח גישה לשרת, אלא כל סיפוק יכולת כלשהיא נקראת שירות.

באנגלולר, אנו ניציר מחלקות אשר מספקות שירותים כדוגמת פונקציות שחזרות על עצמן וכך' באמצעות `.service`.

היתרון הוא קודם כל שהקוד מרוכז במקום אחד ומסודר, יתרון נוסף הוא, שאין צורך לבצע `new` למחלקה זו בכל פעם מחדש כשיצטרכו אותה אלה יוכל לקבל אותה בהזרקה כ' `dependency` `.injection`.

מחלקת service מוגדרת ככל מחלקת אחרת, אולם נוסיף לה את ה `@Injectable` בכך:

נעשה זאת בצורה הכ' בסיסית לדוגמה כך':

```
import { Injectable } from '@angular/core';

@Injectable()
export class TaskService { }
```

מוסיף לה איזושהி פונקציה:

```
@Injectable()
export class TaskService {
  getTasks():Task[] {
    return TASKS;
  }
}
```

במקום שבו נדרש את השירות הנ"ל, לא נעשה:

```
taskService = new TaskService();
```

אלא נזכיר את זה בבניוי של המחלקה בצורה כזו:

```
constructor(private taskService: TaskService) { }
```

כעת, נוכל להשתמש בשירות לאחר שקיבלנו אותו כهزקה בבניוי:

```
this.tasks = this.taskService.getTasks();
```

כעת, מה שעוד נשאר לנו, זה לרשום את הcla `service` הזה כספק שירות. יש להגידו כ provider. במקומות שבו נגדיר אותו כ provider – שם ומטה הוא יוכר, ולכן שירותים כלליים עדיף להגיד ב `app.module`, ואילו שירותים המיועדים לקומפוננטה מסוימת נוכל להגיד כ provider בקומפוננטה עצמה.

צורת הגדרת provider מתבצעת ב metadata של Component או NgModule כרך:

```
providers: [TaskService]
```

provideIn

כאשר אנו רוצים להגדיר מותך service עצמו לתוכו מי הוא יזרק נוכל לעשות זאת באמצעות ה metadata של ה Injectable. ישים מאפיין בשם providedIn אשר יכול לקבל את הערך 'root' שאומר ששירות זה יזרק במודול השורש של האפליקציה, או לתת ערך כלשהו של סוג המחלקה אליה נרצה להזירק אותו.

אפשרות זו מועילה במקרה שאין לנו רוצים סתם להגדיר provider לשירות שלא בהכרח נctrller וחבל על הקוד שיעליה בשל כך, נוכל להגדיר בשירות עצמו – במידה וצרכו אותו – היכן למקם את המופיע שלו.

יתרנו נוסף זה במימוש loading lazy – כאשר אנו כתבים שירות שנמצא במודול שנטען ב lazyอลומ הצורך הוא שירות זה יזרק ב AppModule דוקא – נוכל לבצע זאת בקבלה ע"י providedIn ולא נctrller לוותר על ה lazy בשל כך ולהגדיר אותו מראש ב AppModule.

IMPLEMENTATIONS שונים ל service אחד

ניתן להגדיר בצורה נוספת, במידה וקיים interface עם חתימות, או abstract שיש לו יורשים שונים, נוכל להגדיר את הבסיס ועם איזו מחלקה בפועל להשתמש:

```
providers: [{ provide: TaskService, useClass: TasksServiceHttp }]
```

ה provide מציין מי מחלקת הבסיס, ואילו useClass מפנה למחלקה אותה ישמשו בפועל.

בבנייה של המחלקה הדרוש לא נשנה כלום, נגדיר שאנו ממשיכים לקבל את המחלקה הבסיסית TaskService, אולם בפועל נקבל ליד את המימוש של TasksServiceHttp.

Promise

עד עכšíו הגדרנו את פונקציות כפונקציות שמחזירות ערך מסוים. למשל:

```
const TASKS: Task[] = [...];
getTasks(): Task[] {
    return TASKS;
}
```

אולם, במקרה ומדובר בשלייפה של נתונים רבים שיכולה לארוך הרבה זמן - אנו לא רצאים שהמערכת יכולה תיתקע בשל כך, ולכן נרצה לבצע את הפונקציה כתהיליך אסינכרוני. ממילא, הפונקציה כבר לא תחזיר ערך מיידי ברגע הקראיה אליה, אלא תחזיר הבטחה לאותו ערך – כלומר הערך יחזור אבל לא מיד, יש לנו הבטחה שנתקבלת התשובה לה אנו מצפים.

הצורה בה נגדיר פונקציה כפונקציה שמחזירה הבטחה לערך מסוים שתבצע כך:

```
getTasks(): Promise<Task[]> {
    return Promise.resolve(TASKS);
}
```

במקום הגדרת סוג הערך – נגדיר הבטחה לסוג הערך: promise ל promise. resolve בPromise.resolve שזהו בעצם נקודת המעבר מהמתנה לתשובה – לנטינת התשובה.

הצורה בה נשימוש בפונקציה גם כן תראה מעט שונה, כי הרי בהפעלת הפונקציה – הפעלנו את התהיליך מבלי לקבל עדין תשובה, קיבלנו רק הבטחה לתשובה. علينا אם כן, לכתוב קוד שיחליט מה לעשות במקרה שההתשובה הגיעה. נעשה זאת כך:

```
taskService.getTasks().then(data => this.allTasks = data);
```

ע"י ה then אנו מגדירים בעצם מה יעשה הקוד לאחר קבלת התשובה – מן callback.

ה data אינה מילה שמורה, ניתן לכתוב כל שם שהוא, הסוג של data אותו נקבל ב then יהיה כמובן הסוג שהוגדר ב Promise.

ניתן לחוש את ההבדל ע"י השהיית הקוד ל 5 שניות ולראות איך התהיליךollo לא נעצר בשל כך, המערכת ממשיכה הלאה, וברגע שיש תשובה היא מקבלת אותה ועשו איתה מה שצער.

ניתן גם להגדיר מה יקרה במצב של שגיאה, בה לא נקבל את התשובה. לדוגמה כך:

```
taskService.getTasks().then(data => this.allTasks = data).catch(err=> {
    console.log(err); })
```

להלן קוד המשווה את החזרת התשובה ב 5 שניות:

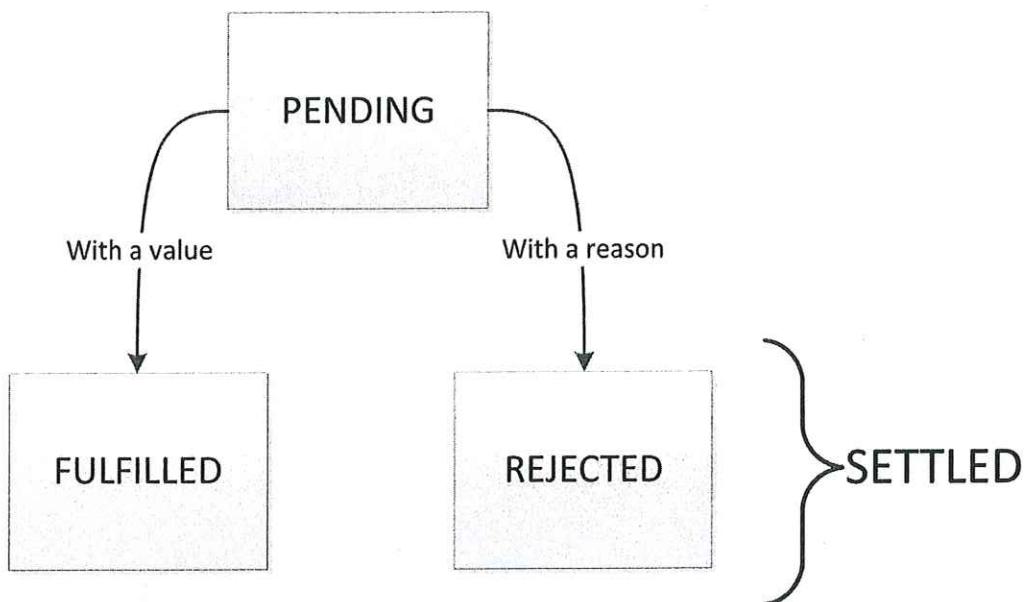
```
getAllTasksSlowly(): Promise<Task[]> {
    return new Promise(res => {
        setTimeout(() => res(TASKS), 5000)
    });
}
```

Angular v.7

new Promise מביא לנו שני ארגומנטים – הראשון זה ה resolve והשני זה ה reject. נשתמש הראשון (res לפי הדוגמה לעיל) בכדי להעביר את ההתנה למצוות של תשובה. ובשני נשתמש בכדי להעביר את ההתנה למצוות של שגיאה, מצוב זה "יתפס עי" מי שיקרא לפונקציה בחלק של ה catch ולא של ה then.

```
new Promise((resolve, reject) => {
  //resolve(123);
  reject(new Error("Something awful happened"));
});
```

ניתן להבין זאת עי' האירור הבא:



ניתן להרחיב עוד רבות על עניין ה promise מומלץ ללמידה את הנושא לעומקו.

Http Client

HttpClientModule

אנגלר הcin עברנו מודול המטפל בכל הקשור לתקשורת בין client לserver. המודול אליו יש לעבוד נקרא HttpClientModule.

ניבא אותו ונצהיר עליו במודול הראשי:

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [BrowserModule, FormsModule, HttpClientModule],
  declarations: [AppComponent, TasksComponent, TaskDetails],
  bootstrap: [AppComponent],
  providers: [TaskService]
})
export class AppModule { }
```

כאשר נרצה להשתמש עם המודול, נצטרך להזמין אותו במבנה של אותה המחלקה. נעשה זאת כך:

```
import { Injectable } from "@angular/core"
import { HttpClient } from '@angular/common/http'

@Injectable()
export class TasksHttpService {
  constructor(private http: HttpClient) { }

}
```

במעבר מהיר על אפשרות המודול נוכל לראות כי יש בו את כל סוגי בקשות הגישה לשרת: get, post, put, delete וכו'.

כל הפונקציות הללו מוחזירות Observable – מה זה Observable

Observable

Observable הוא קוד שנכתב ע"י ספרית RxJS (או ס_xs שזו הספרייה של עבר קוד SJ) ספריה זו מיועדת עבר אסינכרונית של קוד.

נתאר זאת ע"י ציר זמן, שבו יכול להגיע מידע אחד או יותר, ציר הזמן יקרא data-stream וכל מידע שיגיע יקרא pulse.
לדוגמה: זרימת מידע data-stream אודות לחיצה על כפתור במסך, זהו בעצם ציר זמן בו המספר ח'י, בו ישנו כפתור אשר בכל לחיצה עלי' נרצה לבצע משווה. כל לחיצה תהווה pulse של מידע בקצב המידע הכללי, רצף מידע זה (data-stream) יכול להיות שיופיע בו pulse אחד או יותר, או אף אחד, זמן וכמות ה pulse אינם ידוע מראש.

אם הבנו את הרעיון הזה – נוכל להבין מה צריך בו כאשר מתבצעת קריאה לשרת. כאשר מתבצעת קריאה לשרת – נפתח בעצם ציר זמן כלשהו, ובו אנו מחכים לתשובה שתגיעמן

השרת – אנו לא יודעים מתי התשובה תגיע.
בקראות לשרת – ישנו ציר מידע ובו פולס אחד בלבד.

ונכל להשווות זאת לצורה בה קראנו ל `ajax` וכתבנו callbacks עבור `success` ועבור `error`.

באותה מידה גם כאן, כל גישה לשרת תחזיר מתי שהוא והוא יכולה להצליח או להכשל, במידה והגישה הצלחה אנו נחזר נ נתונים מסוימים (את סוג הנתונים לו נצפה), במידה והגישה תכשל, מכל סיבה שהיא – התשובה לא תהיה הרו סוג הנתונים להם נצפה – מכיוון שהם פשוט לא יתהפכו – הגישה נכשלה. אנו נקבל במקרה של כישלון את ה `response` ונוכל לנתח אותו ולדעת מה קרה ולמה.

מה ההבדל בין Observable לבין Promise ?

מה ההבדל בין Observable לבין Promise ? שניים לכוארו אסינכרוניים וממשים הבטחה להחזרת נתונים מסוימים.
ישנים הבדלים גדולים מאד, אולם נתמקד בראשון:

Promise לעולם ייחזר תשובה אחת. ככלומר התשובה תמיד אחת ותמיד תהיה מסווג הנתונים אותו הגדכנו. ציר המידע יכול `pulse` אחד מסווג אחד.

לעומת זאת, Observable, משככל את `Promise`, ומגדיר ציר זרימת מידע ובו אפשרות אינסופית של פולסים מסווגים זהים או שונים.

`get`

התוצאה של קראת `get` או כל קראיה אחרת תהיה מסווג `Object` או סוגים נוספים
אחרים, אולם ככלם יהיו `T`.

אם נרצה להציג בשירות שלנו סוג נתונים מסוימים – נדרש למפות את אובייקט `response` ולהציגו
אותו בצורה אליה אנו מצפים נעשה זאת ע"י שילוח ה `T` למתודה `get` בצורה זו:

```
getAllTasksFromServer(): Observable<Task[]> {
  return this.http.get<Task[]>("api/Tasks");
}
```

הצורה בה נקרא לשירות ונקבל ממנו את הנתונים לאחר שביצע את הגישה לשרת ת策רף כתיבה
מיוחדת. זאת מכיוון, שאנו מפעילים כאן בעצם תהליך אסינכרוני שבמסוף של דבר מגיע לתשובה
כלהיא, התשובה לא מגיעה באותו הרגע בה הפעלו את התהליך, ולכן עלינו להציג מזמן כלשהו,
שיחכה למידע שיחזור מן התהליך ויפעל לפיו בהתאם.

המבחן הוא בעצם הצמדת ה `subscribe` לקראיה:

```
ngOnInit() {
```

Angular v.7

```
        this.tasksService.getAllTaskHttp().subscribe(tasks => this.allTasks =
      tasks, err => { console.log(err); });
    }
```

ההازנה לזו מתבצעת באמצעות subscribe. ה subscribe יძין לפולסים שייגעו והוא יכול לקבל 3 סוגים פרמטרים:

הראשון – מקרה של פולס מידע תקין – מבחיננו זו תהיה תגובה הצלחה של השירות – אנו נכתוב קוד מה יקרה במקרה של הצלחה, ניתן לראות כי ה data אותו נקבל יהיה מן הסוג אותו הגדרנו Observable של השירות:

```
getAllTasksFromServer(): Observable<Task[]>
```

הפרמטר השני יהיה הכישלון – אנו נכתוב קוד מה לעשות במקרה של כישלון. זהו פולס מידע אודות כישלון ברגע המידע, רצף המידע יסגר ע"י כישלון זה.

השלישי הוא complete – סוג של callback לאחר השלמת התהליך – לא מקבלים נתונים כלשהם, בלבד אם אין להזיה שימוש.

הפעלת השירות ללא subscribe תגרום לקריאתו ולהשארתו "קפוא". רק ה subscribe משחרר את התשובות למעשה.

שליחת פרמטרים בקריאה Get

אם נרצה לשלוח פרמטרים בקריאה get יהיה علينا להוסיף queryString את הפרמטרים.

נעשה זאת כך:

```
return this.http.get("api/Tasks/Get?active=true")
```

שליחת נתונים בקריאה Post

בקראית post אנו יכולים לשלוח אובייקטים מורכבים בצורה פשוטה ביותר:

```
saveAllTasks(tasksToSave: Task[]): Observable<boolean> {
  return this.http.post("/api/Tasks", tasksToSave).map(res => { return
  true; }).catch(err => { return Observable.throw(err); });
}
```

השתמשו בקריאה post ושלחו לפרמטר body את הנתונים שלנו, בדוגמה זהה רשימה של שירותי.

שינוי הגדירות *Request*

כאשר אנו רוצים לשנות הגדירות או להוסיף על נתוני `request` עליון לשלוח את ההגדירות הללו לפרמטר האופציונאלי `skype` בכל פונקציה בקיט `http` בשם `options`.

אובייקט זה הוא מסוג `HttpHeaders` ו לשם כך ניבא מחלוקת זו:

```
import { HttpHeaders } from '@angular/common/http';
```

את ההגדירות שנרצה להציג נכתב כך לדוגמה:

```
const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

ונשלח אותו לפרמטר `options` כך:

```
saveAllTasks(tasksToSave: Task[]): Observable<boolean> {
  return this.http.post<boolean>('/api/Tasks', tasksToSave, httpOptions);
}
```

Async pipe

כאשר אנו מציגים מידע אסינכרוני, ישן כמה דרכים לעשות זאת.

הדרך הראשונה היא להציג את המידע כמאפיין בעל ערך רגיל, להציג מАЗין `observable` וכאשר מגיע המידע – לשפוך אותו לתוך המאפיין. התצוגה תשתנה בהתאם.

בקוד ה `ts` נכתב כך:

```
class AsyncPipeComponent {
  observableData: number;
  subscription: Subscription = null;

  constructor() {
    this.subscription = this.getObservable()
      .subscribe(data => this.observableData = data);
  }

  getObservable() {
    return Observable
      .interval(1000)
      .take(10)
      .map((v) => v * v);
  }
}
```

Angular v.7

```
ngOnDestroy() {
  if (this.subscription) {
    this.subscription.unsubscribe();
  }
}
```

בtemplate נכתב כך:

```
<p>{{ observableData }}</p>
```

בדרכו הזו ישנו חיסרונו כיון שהוא צריך לכתוב את ה subscribe וולדואג במידת הצורך לעשות גם unsubscribe.

ישנה דרך שנייה, והיא להגדיר את המאפיין observable obs ואז להציגו ב template עם pipe בשם async. (פירוט בנושא pipes נמצא בהמשך).

Pipe זה מצמיד באופן אוטומטי מזמן למאפיין זהה ודואג גם לסגור אותו.

כל זה נכון גם ל promise (אםນם אין את עניין הסגירה, אולי הת יבור זהה | |

בצורה הזו הקוד נקי יותר ונכון יותר.

בקוד ה dz נכתב כך:

```
class AsyncPipeComponent {
  observableData: Observable<number>;
  constructor() {
    this.observableData = this.getObservable();
  }
  getObservable() {
    return Observable
      .interval(1000)
      .take(10)
      .map((v) => v*v)
  }
}
```

בtemplate נכתב כך:

```
<p>{{ observableData | async }}</p>
```

Observables operators

ספרית **ReactiveX** יוצרה עשרה אופרטורים "יעודים" לפעולות מוכרות ונוצרות.

האופרטורים מתחלקים לקטגוריות, למשל אופרטורים המיעדים ליצור **data-stream**, אופרטורים המיעדים לסתון, פילטור, טרנספורמציה ועוד'.

השימוש באופרטורים מתאפשר באמצעות יבוא שלהם דרך import וכל אופרטור יש את צורת השימוש שלו.

להלן דוגמה לשימוש באופרטור from שהוא אופרטור המיעד ליצור observable ואופרטור map המיעד למיפוי, אופרטור זה יקבל את ה **data** ויחזר במקומו **data** אחר לפי מה שנרשום לו.

בדוגמה להלן ניצור **data-stream** מתוך מערך מספרים, זה אומר שהמספרים הללו יגיבו כפולסים של מידע ברצף שייפתח ע"י האזנה אליו. ה **data-stream** הזה יכנס לתוך המשתנה **source**.
לאחר מכן, נבצע מיפוי ע"י האופרטור **map**, אופרטור זה יוכנס תחת **map**, הוצאה ההז אומرتה ש**observable** הינו קיים, אולם בדרכו החוצה עליו לעבור מספר שינויים. בדוגמה שאכן, ה **Observable** יצטרך לעבור מיפוי לפני צאתו. המיפוי ישנה כל פולס מידע להיות הערך המקורי+10.
הازנה ל **observable** זה (**subscribe**) תמחיש לנו את התוצאה.

```
import { from } from 'rxjs';
import { map } from 'rxjs/operators';

//emit (1,2,3,4,5)
const source = from([1, 2, 3, 4, 5]);
//add 10 to each value
const example = source.pipe(map(val => val + 10));
//output: 11,12,13,14,15
const subscribe = example.subscribe(val => console.log(val));
```

בדומה לדוגמה זו, ניתן להבין ולצרוך אופרטורים שימושיים ביותר מתוך עשרה כאלה.

ניתן למצוא אותם באתר של **ReactiveX** : <http://reactivex.io>

או באתר דוגמאות זה : <https://www.learnrxjs.io/operators>

Routing - Navigation

RouterModule

אנגולר מכיל מודול המטפל בכל נושא הניות באפליקציה. מודול זה נקרא `RouterModule`.
קודם כל علينا לייבא אותו ולהגדיר לו את מערכת הניותים באפליקציה:
ייבא אותו ונצהיר עליו במודול הראשי:

```
import { RouterModule, Routes } from "@angular/router"

@NgModule({
  imports: [BrowserModule, FormsModule, HttpClientModule, RouterModule],
  declarations: [...],
  bootstrap: [AppComponent],
  providers: [...]
})
export class AppModule { }
```

הצהרה זו אינה מספקת, כיוון שאנחנו לא מודול את הנוטים הקיימים במערכת.

Routes

```
const ROUTES: Routes = [
  { path: "tasks", component: TasksComponent },
  { path: "users", component: AllUsersComponent },
  { path: "", component: HomePageComponent },
  { path: "**", component: PageNotFoundComponent },
];
```

מערך זה מגדיר עבור כל מילת ניוט `path` מהי הקומפוננטה אותה נרצה להציג. כלומר, כאשר הניות יהיה ל `tasks` נרצה לראות את הקומפוננטה של רשימת המשימות וכן הלאה.

הגדרת `path` ריק יווה הגדרה לדף בירת מחדל, דף ראשי כלשהו.

הגדרת `path` של `**` מבהו בעצם קליטה של כל מה שלא הוגדר עד כה. כלומר כל ניוט אחר לא מוכר יגיע לדף 404.

נצרף את המערך למודול כך:

```
@NgModule({
  imports: [BrowserModule, FormsModule, HttpClientModule,
    RouterModule.forRoot(ROUTES)],
  declarations: [...],
  bootstrap: [AppComponent],
  providers: [...]
})
export class AppModule { }
```

כעת, علينا להגדיר תפריט ניוט כלשהו ומשטח טעינה דינامي אליו יטען בהתאם הקומפוננטות הרצויות לפני הניות.

Angular v.7

ב `app.component.html` נגידר תפריט פשוט, אולם במקומם הקישורים הרגילים נשתמש בעיצובה directive `routerLink` שנקרא

```
<nav>
  <ul class="nav nav-tabs">
    <li routerLink="/tasks" routerLinkActive="active"><a>All Tasks</a></li>
    <li routerLink="/users" routerLinkActive="active"><a>All Users</a></li>
  </ul>
</nav>

<router-outlet></router-outlet>
```

ה `routerLinkActive` הינו directive שמקביל שם של `class` ב `css` ומציב אותו כאשר הקישור הנוכחי פעיל. זו אינה מילה שמורה, זו בסה"כ שם של מחלוקת בbootstrap.

`router-outlet` זהו בעצם משטח הטעינה הדינמי, אליו יטענו התכנים בהתאם לניווט שיקש.

דבר אחרון שנדר לעשות זה לשים תגית `<base href="/">` תחת `<head>` בדף הראשי (`index.html`)

תגית זו אומרת למנוע הניווט מי היא כתובות הבסיס ממנה והלאה הוא צריך לפענה את ה `url`.

ניווט ע"י קוו

```
<ul>
  <li *ngFor="let user of users" (click)="selectUser(user)">
    <a>{{user.name}}</a>
    {{user.name}}
  </li>
</ul>

import { Component } from "@angular/core"
import { USERS, user } from "../../models/task"
import { Router } from "@angular/router"

@Component({
  selector: "all-users",
  moduleId: module.id,
  templateUrl: "all-users.component.html"
})
export class AllUsersComponent {
  constructor(private router: Router) {
  }

  users = USERS;

  selectUser(user: user) {
    this.router.navigate(['/tasks', { user: user.id }]);
  }
}
```

Angular v.7

חשיבות הניווט זה-מתבצע באמצעות מנגנון הניווט של RouterModule ולא דרך הניווט של הדף כפשותו, ולכן ניתן כתובות כלשהו שלא דרך המודול (לא דרך router.navigate ולא דרך directive (routerLink directive) הדף לא באמת ידע לאן לgesture ונקבל שגיאת 404 של ניווט לא קיים.

בשביל לפתור בעיה זו יש כמה דרכיים, הראשון פשוטה ביותר היא לגרום כתובות להיחזות לשני חלקים: החלק האמיטי שגם הדף יידע לgesture אליו, והחלק השני שהוא הניווט שייתבצע באמצעות אングולר. הפרדת החלקים תהיה ע"י הסימן #.

נעשה זאת ע"י הגדרה פשוטה ב import של המודול:

```
RouterModule.forRoot(ROUTES, { useHash: true })
```

ישנה דרך נוספת, והוא ע"י שימוש במחלקה מיוחדת שנוצרה לשם כך:

ניבא אותה כרך:

```
import { HashLocationStrategy, LocationStrategy } from "@angular/common"
```

ונגדיר אותה באמצעות provider מיוחד:

```
providers: { provide: LocationStrategy, useClass: HashLocationStrategy }
```

ניווט באמצעות פרמטרים

כאשר נרצה לשלוח פרמטרים לקומפוננטה נוכל לשלוח זאת באמצעות הניווט. אם בחרנו לנות דרך html נוכל לשרשר לכתוב את הפרמטר בצורה זהה לשירותו וגיל של פרמטרים בדף.

לדוגמה:

```
<a routerLink="/tasks?user={{user.id}}">{{user.name}}</a>
```

או:

```
<a routerLink="/tasks/{{user.id}}">{{user.name}}</a>
```

יש להוסיף במערך הניווטים ניווט המיועד לפרמטרים בכך שהמנגן ידע להתייחס לזה ולא לקרוא.

```
const ROUTES: Routes = [
  ...
  { path: "tasks", component: TasksComponent },
  { path: "tasks/:user", component: TasksComponent },
  ...
];
```

Angular v.7

בשני המקרים הפורמט שנשלח יכנס למשתנה user מכיוון שבצורה הראשונה ממש כתבנו זאת ידנית ע"י queryString ובצורה השנייה שלחנו פורמט אחד בלבד מה שאומר שזה יכנס לפורמט בירית מחדל שהוגדר עבור הקישור זהה.

אם בחרנו להשתמש בנינוי דרך קוד, נוכל לשלוח את הפורמטרים בצורה כזו:

```
this.router.navigate(['/tasks', { user: user.id }]);
```

או:

```
this.router.navigate(['/tasks', user.id]);
```

עכשו, כל שנוטר הוא קלוט את הפורמט שנשלח ולחוץ ממנו את הערך.

לשם כך אングולר הקצתה אובייקט בשם ActivatedRoute המכיל מידע בזמן אמת אודות הנינויים במערכת.

לצורך קליטת ערך הפורמט שנשלח עליינו להקצות בקומפוננטהamazon, שיאזין לכל שינוי שעובר על אובייקט זה.amazon זה יתריע עבורנו על שינוי בפורמטרים ונוכל לחשאל את הפורמטים המסויים אותו אנו מבקשים - במידה ויהי קיימן - נשלוף את ערכו ונעשה בו שימוש.

חשוב לציין שאובייקט זה הוא אובייקט אחד עבר כל המערכת שאנו מקבלים אותו בהזרקה, ולכן כאשר נצמיד אליו amazon,amazon ימשיך לחיות גם לאחר הריסת הקומפוננטה והוא יכול לגרום לאליגט זיכרון לא רציה, כך שחשוב מאד לבטל את ההאזנה בסיום חייה הקומפוננטה בכך למנוע מקרה זה.

הצמדת amazon תהיה בצורה כזו:

הגדרת subscription

```
export class TasksComponent implements OnInit, OnDestroy {
    ...
    subscription: Subscription;
    ngOnInit() {
        this.subscription = this.route.params.subscribe((params: any) => {
            this.tasksHttpService.getAllTasksFromServer().subscribe(data => {
                if (params['user'] != undefined) {
                    this.allTasks = data.filter(function (e, i) {
                        return e.userId == +params['user'];
                    })
                }
                else this.allTasks = data;
            });
        });
    }
}
```

ביטול ההאזנה יעשה כך:

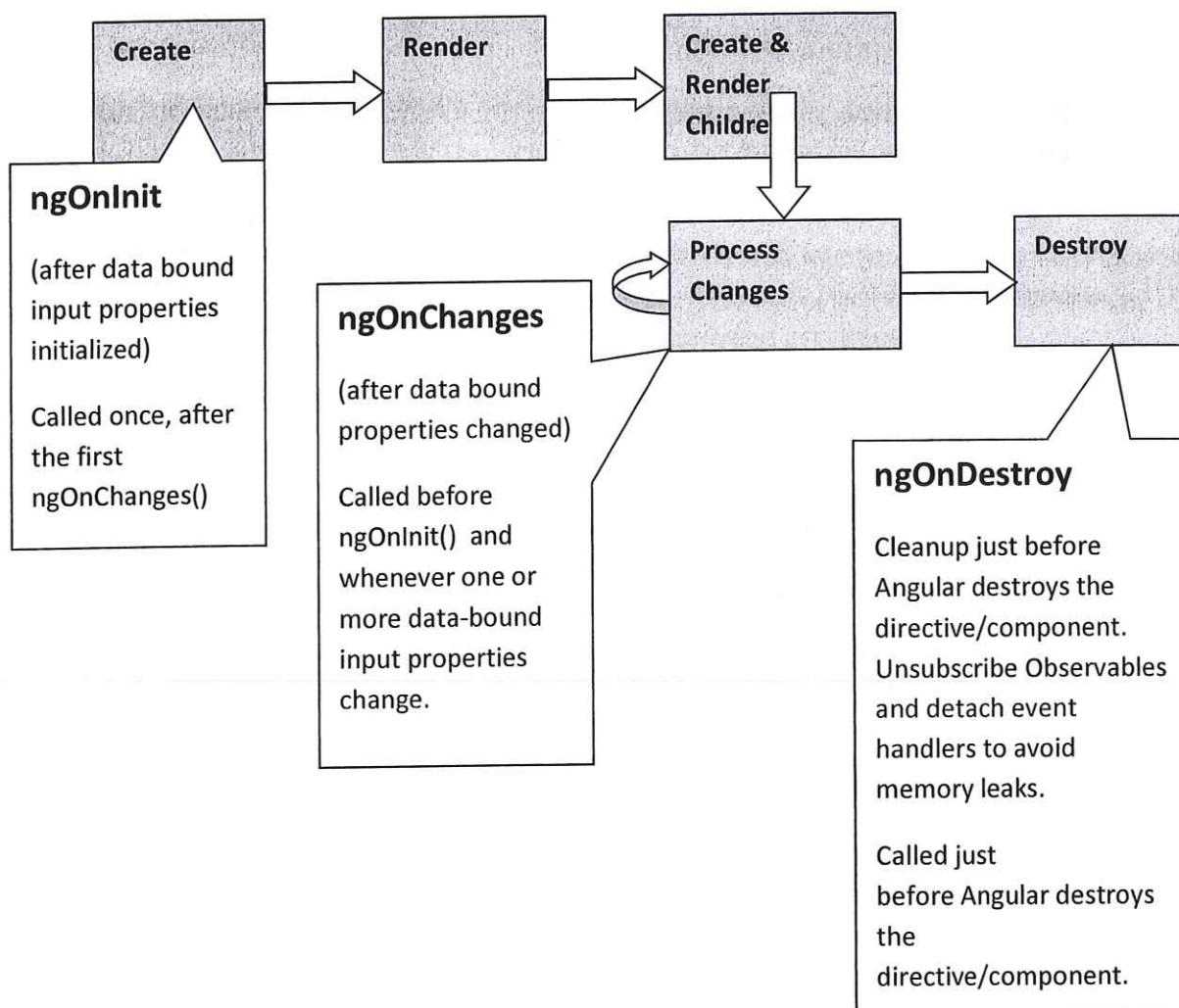
```
ngOnDestroy() {
    this.subscription.unsubscribe();
}
```

LifeCycle Hooks

מחזור חי' הקומפוננטה מורכב מכמה שלבים:

הקומפוננטה נוצרת ומתרנדרת, יוצרת ומרנדרת את הילדים תחתיה, בודקת האם בוצעו שינויים במאפיינים והורסת את הקומפוננטה לפני שהיא מוסרת מן ה-DOM.

בתרשים הבא מתואר מחזור חי' הקומפוננטה, האירועים הראשיים הם ngOnInit, ngOnChanges, Process Changes וngOnDestroy. הסבר ופירוט מטה:



תקשורות בין קומponentות Component Interaction

תקשורות בין קומponentות זהו נושא חשוב מאד, כיוון שצורת הפיתוח באנגולר מאופיינת בכתיבת רכיבים סגורים, אשר יש צורך לתקשר ביניהם בהעברת מידע או בהוצאה.

ישנן כמה דרכים לתקשורות בין קומponentות:

Input - Output

Input זהה האפשרות לקלוט לתוך הקומponentה ערכים מבחן – ניתן לשימוש בתקשורות ביןABA לבן.

Output זהה האפשרות להוציא פלט מתוך הקומponentה החוצה – ניתן לשימוש בתקשורות בין הבן לאבא.

פירוט נרחב ניתן למצוא לעיל בנושא זה.

דוגמאות ל:

ה **task** יוגדר כך:

```
@Component({
  selector: "task-details",
  templateUrl: "task-details.component.html",
})
export class TaskDetails {
  @Input()
  task: Task;
}
```

שליחת הנתונים ל **task** תבוצע כך:

```
<task-details [task]="selectedTask"></task-details>
```

דוגמאות ל **output**:

ה **onSaveTask** יוגדר ויעפעל כך:

```
@Component({
  selector: "task-details",
  template: `<button (click)="saveTask()"></button>`,
})
export class TaskDetails {
  @Input()
  task: Task;

  @Output()
  onSaveTask: EventEmitter<Task> = new EventEmitter<Task>();

  saveTask() {
    this.onSaveTask.emit(this.task);
```

Angular v.7

```
}
```

ההרשמה לפלא זה תבוצע כך:

```
<task-details [task]="selectedTask"
(onSaveTask)="saveTaskToList($event)"></task-details>
```

Template Reference Variable

האפשרות ליצור משתנה ב template על אלמנט יכולת להתבטה גם ביצירת משתנה על קומפוננטה. בדרך זו נקבל לתוך המשתנה את כל תכונות הקומפוננטה ונוכל להפעיל דרך מethod או לגשת למאפיינים. ניתן לשימוש בתקשרות בין אב לבן.

לדוגמה, אם בקומפוננטת task-details ישנה מmethod בשם doSomething, נוכל להפעיל אותה כך:

```
<task-details #taskDetails></task-details>
<button (click)="taskDetails.doSomething()"></button>
```

Service

אפשרות מצינית לתקשרות בין קומפוננטות ללא תלות של קשרי אב-בן היא באמצעות service. service מזמין לכל מחלוקת בה הוא מתבקש ואני לאחxon מידע מסווג ב service וכך להגיע למידע זה מכל קומפוננטה.

פירוט נרחב ניתן למצוא לעיל בנושא [Services](#).

ViewChild – ContentChild

ViewChild

כאשר נרצה לגשת למאפיינים או מmethods של קומפוננטה כלשהיא מתוך הקוד במחלוקת, לא נוכל להסתפק ב [template reference variable](#).

Angular v.7

ונכל לבצע זאת באמצעות הגדרת `ViewChild`: decorator. אפשרות זו נותנת לנו להגדיר משתנה שיקבל אליו את המחלקה של הקומponeנטה הרצiosa.

הHIPOSH אחר הקומponeנטה הרצiosa יבוצע לפי מה שנגידר.

אפשרות אחרת היא להגדיר את סוג המחלקה אותה אנו מחפשים.

לדוגמא, אנו מחפשים קומponeנטה שהמחלקה שלה היא `ColorSampleComponent`, נגידר `ViewChild` וניתן לו כפרמטר את ה `type` של המחלקה, המשתנה עצמו יהיה מסוג אותה מחלקה:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  ...
  ...
  @ViewChild(ColorSampleComponent)
  primarySampleComponent: ColorSampleComponent;
  ...
}
```

מתי נוכל להתחיל להשתמש בתוינו המחלקה? רק לאחר מציאתה ורינדורה. נקודת זמן זו ניתנת לנו ע"י שימוש ה `AfterViewInit`

חשוב להבין, קודם כל מתרנדורת הקומponeנטה אב, ורק לאחר מכן מתרנדורות קומponeנטות הילדיים שבתוכן. כך שאירוע `OnInit` בקומponeנטה האב מתיחס רק לאייחול של קומponeנטת האב ולא בהכרח קומponeנטת הבן מוכנה כבר לשימוש. במידה ורוצים לוודא קומponeנטת הבן גם כן נטענה, נוכל לבצע את הקוד המתיחס לקומponeנטה הבן רק בעת אירוע `ngAfterViewInit`.

לדוגמא:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements AfterViewInit {
  ...
  ...
  @ViewChild(ColorSampleComponent)
  primarySampleComponent: ColorSampleComponent;

  ngAfterViewInit() {
    console.log("primaryColorSample:", this.primarySampleComponent);
  }
  ...
}
```

ונכל למצוא קומponeנטות או אלמנטים גם לפי שם של משתנה שנמצמיד אליו ב `template` ע"י `template reference variable`. את השם זהה ניתן כפרמטר ל `ViewChild` במקום סוג המחלקה.

לדוגמא, נמציד משתנה `tr` בשם `title` לאלמנט `h2`:

Angular v.7

```
<h2 #title>Choose Brand Colors:</h2>
```

ונכל לחפש אלמנט זה ולהתיחס אליו בקוד כך:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent implements AfterViewInit {
  .....
  @ViewChild('title')
  title: ElementRef;
  ngAfterViewInit() {
    console.log("title:", this.title.nativeElement);
  }
  .....
}
```

ViewChildren

decorator זה נותן תוצאה של כל מופע הילדים שנמצאו בשונה מ ViewChild שנตอน את המופיע הראשון שנמצא. עובד בצורה זהה ל ViewChild או למחריז QueryList.

ng-content

כאשר צורכים קומפוננטה – יש לשים את תגיית הקומפוננטה שהוגדרה לפי selector שלה.

לדוגמה הוגדרה קומפוננטה המקבלת טקסט כלשהו ומציגו באותו פאנל מיועד:

```
@Component({
  selector: 'app-panel',
  templateUrl: '<p class="panel">{{text}}</p>'
})
export class AppPanelComponent {
  .....
  @Input()
  text: string;
}
```

צורת השימוש בקומפוננטה זו תהיה כך:

```
<app-panel [text]="'text to show'"></app-panel>
```

צורה זו מאפשרת קלילות טקסט בלבד.

כאשר נרצה לאפשר לקלוט תוכן שאינו רק טקסט אלא html המכיל תגיות שונות ואפיו קומפוננטות נוספות וזאת מבייש לשולח זאת כפרמטר למשתנה `shout` כלשהו (שיקבל את זה בצורה לא טוחה כמו

Angular v.7

מחרוזת טקסט) נוכל לתת אפשרות לכותב כל תוכן שהוא בין התייחסות הפותחת של הקומפוננטה לתגיית הסוגרת שלה ולגשת לתוך זה בקבוקות ע"י `ng-content`.

אם ניקח את הדוגמה הקודמת ונשנה אותה לעבוד עם `ng-content` היא תיראה כך:

```
@Component({
  selector: 'app-panel',
  templateUrl: `<p class="panel">
    <ng-content></ng-content>
  </p>`
})
export class AppPanelComponent {
  ....
}
```

צורת השימוש תהיה כuta כך:

```
<app-panel>
  <h1>my first panel</h1>
  <a href="#">link</a>
</app-panel>
```

select

אם נרצה להציג כמה מקומות המאפשרים קליטת תוכן, נוכל להגדיר כמה `ng-content` ולהוסיף להם הגדרת `select`, הגדרה זו אומרת שכאשר בין תגיית הקומפוננטה ישTEL אלמנט עם הסלקטור המוגדר – רק אלמנט זה ישTEL בתוך ה `ng-content` זהה.

לדוגמה, אם נרצה לאפשר שתילת כפתורי חזר והמשך, נוסיף `select` כך:

```
@Component({
  selector: 'person',
  template: `<div class="card card-block">
    I want the prev button in me
    <ng-content select=".prev"></ng-content>
    I want the next button in me
    <ng-content select=".next"></ng-content>
  </div>
`)
```

לכל `ng-content` בקומפוננטה `person` יכנס רק אלמנט מתאים ל `class` שהוגדר (`next` או `prev`)

```
@Component({
  selector: 'app',
  template: `
    <person>
      <button class="next">next</button>
      <button class="prev">prev</button>
    </person>
`)
```

```
}
```

ישנם סוגים נוספים ולאו דווקא דרך שם של קלאס. אפשר לבדוק כמו בcss, לבחור את האלמנט שהוא מסוג `button`, שיש עליו attribute מסוים וכך'.

ContentChild

כאשר נרצה לגשת לתוך כלשהו שנטענו ע"י `ng-content` נצטרך לבצע זאת ע"י הוראות `ContentChild`.

אינו תופס אלמנטים שהוגדרו בתוך `.ng-content`.
לעומתו, תופס רק אלמנטים שהוגדרו בתוך `.ng-content`.
חיפוש הקומפוננטה יבוצע כמו ב `ViewChild` – או לפי סוג המחלקה או לפי שם משתנה `trv`.
הקומפוננטה תהיה זמינה לשימוש רק לאחר `AfterContentInit`.

לדוגמה בקומפוננטת הפאנל, יבוצע חיפוש על אלמנט עם שם link שיוגדר בתוכן שלו:

```
@Component({
  selector: 'app-panel',
  templateUrl: `<p class="panel">
    <ng-content></ng-content>
  </p>`
})
export class AppPanelComponent implements AfterContentInit {
  .....
  @ContentChild('link')
  link: ElementRef;

  ngAfterContentInit() {
    this.link.nativeElement.style.backgroundColor = "red";
  }
}
```

על אותו רעיון קיימים גם:

ContentChildren

זה מוחזיר את כל המופעים התואמים לחיפוש בשונה מ `ContentChild` שמחזיר את המופיע הראשון שנמצא.

Directives

directive זהה בעצם מחלקת אשר נועדה לעזור בחילול קוד התצוגה.
לאנגולר יש directives מובנים, כאשר כמובן, ניתן ליצור כלו בצורה עצמאית לכל מטרה.

Directives מובנים

בפרק זה נעבור על חלק מן directives המובנים ולאחר מכן נציג נציג directive בלבד.
Directive אשר יתחל ב * מסמן על כך שהוא מבצע שינויים ב DOM (מוסיף אלמנטים/מוריד וכו').
חשוב!!! לא ניתן למשוך על אותה תגית שני directives שהם בעלי כוכבית כגון: `ngFor` עם `if`.
במקרה הצורך, ניתן תגית נוספת פנימית יותר שתכילה את ה directive השני הרצוי.

*ngIf

האלמנט שעליו הוא יוגדר יתווסף ל DOM או יוסר ממנו לפי הביטוי שנכתב. לדוגמה:

```
<div *ngIf="show">Text to show</div>
```

הוספה else:

יש להוסיף אלמנט מסווג `ng-template` שם יוגדר האלמנט שיופיע במקרה שה `else` מתקיים.
אלמנט זה, יש להוסיף מזהה כלשהו עם קידומת סולמית.

לדוגמה:

```
<div *ngIf="number==5; else showDiff">
    The number is 5
</div>
<ng-template #showDiff>
    The number is not 5
</ng-template>
```

במקרה זה, נוסף ה`id` `#showDiff` שאינה מילה שומרה ל`ng-template` ובמקרה והתנאי לא מתקיים
יוצג מה שבתוך ה `ng-template` במקום אותו אלמנט שעליו הוגדר התנאי.

*ngFor

האלמנט עליו הוא יוגדר יוצר עבור כל מופיע במערך/אובייקט איטרציוני שנכתב בביטוי. לדוגמה:

```
<ul>
    <li *ngFor="let item of itemsList"> {{ item }} </li>
</ul>
```

Angular v.7

ngClass

האלמנט עליו הוא יוגדר יקבל סט של מחלקות CSS בהתאם לביטויים אותם נרשום. אם הביטויים יჩירו true הattribut class יתווסף לאלמנט, במידה ולא – הattribut class לא יתווסף לאלמנט.

לדוגמה:

```
<div class="panel" [ngClass]="{'collapsed': state!='in', 'main': mainPanel }"></div>
```

ngStyle

האלמנט עליו הוא יוגדר יקבל הגדרות style לפי הביטויים אותם נרשום. לדוגמה:

```
<div [ngStyle]="{'background-color': person.country === 'UK' ? 'green' : 'red' }"></div>
```

ngSwitch, ngSwitchCase, ngSwitchDefault

שלושתם משמשים יחד בצדלי יצור תנאי switch על הצגת אלמנטים לפי הביטוי שנכתב.

לדוגמה:

```
<ul *ngFor="let person of people"
    [ngSwitch]="person.country">

    <li *ngSwitchCase="'UK'"
        class="text-success">{{ person.name }} ({{ person.country }})

    </li>
    <li *ngSwitchCase="'USA'"
        class="text-primary">{{ person.name }} ({{ person.country }})

    </li>
    <li *ngSwitchCase="'HK'"
        class="text-danger">{{ person.name }} ({{ person.country }})

    </li>
    <li *ngSwitchDefault
        class="text-warning">{{ person.name }} ({{ person.country }})

    </li>
</ul>
```

ngModel

האלמנט עליו יוגדר יבצע two way binding למאפיין אותו נרשום, הרחבה ניתנת לקרוא בנושא הטפסים. לדוגמה:

```
<input type="text" name="first-name" id="first-name" [(ngModel)]="firstName" />
```

ngForm

המשתנה אליו הוא יושם המוגדר על אלמנט מסווג טופס - מקבל את כל תכונות ומאפייני הטופס אוטומטiquement מכאן עבור העבודה עם הטפסים. הרחבה ניתנת לקרוא במשא הטפסים. לדוגמה:

```
<form #frmSearch="ngForm"> ... </form>
```

routerLink

בלחיצה על האלמנט עליו הוא מוגדר יתרבצע ניוט לכתובת שנרשום. לדוגמה:

```
<ul class="nav nav-tabs">
  <li routerLink="/tasks"><a>All Tasks</a></li>
  <li routerLink="/users"><a>All Users</a></li>
</ul>
```

routerLinkActive

האלמנט עליו הוא יוגדר יקבל את המחלקה CSS שנרשום כאשר הניווט הנוכחי במערכת מצביע לנישוט שלו מוגדר ה routerLink באוטו אלמנט. לדוגמה:

```
<ul class="nav nav-tabs">
  <li routerLink="/tasks" routerLinkActive="active"><a>All Tasks</a></li>
  <li routerLink="/users" routerLinkActive="active"><a>All Users</a></li>
</ul>
```

router-outlet

האלמנט עליו הוא יוגדר ישמש כמשטח טעינה דינמי אליו יטמע הקומפוננטות לפי הגדרות הנישוטים. לדוגמה:

```
<router-outlet></router-outlet>
```

Custom directive

כפי שהזכר directive זהה מחלוקת, אנו נוסיף עלייה metadata והתנהגות תואמת.

ניתן לבנות באופן עצמאי ונitin גם להיעזר בו זו וליצור דרכו בקלות directive חדש.

בדוגמה זו נבנה directive הצובע אלמנט עליו הוא מוגדר לפי הצבע המבוקש.

לdirective נקרא highlight ואם נרים את הפיקודה:

Angular v.7

ng generate directive highlight

נקבל קובץ חדש בשם highlight.directive.ts ודבר חשוב נוסף – הערה עליון במודול במבנה ה declarations.

הקוד שנתקבל יהיה זהה:

```
import { Directive } from '@angular/core';
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

הוגדרה כאן מחלוקת בשם `HighlightDirective` מחלוקת זו ריקה כרגע. למחלוקת זו הוגדר מסוג `@Directive`_decorator שקיביל metadata ומו מידע על הסלקטור עליו יתפוסף directive הינה directive הינה – הפעלתו הינה אומר שכל אלמנט שיוגדר עליו directive והוא תואם לselector הניתן – הפעלתו הינה – הפעלתו. במקרה זה הסלקטור הוא אטሪיבוט בשם `appHighlight`.

elementRef

כאשר נרצה לבצע שינוי כלשהו באלמנט – נוכל לקבל את האלמנט עצמו בהזירה לתוך הבניין, האלמנט עצמו יתקבל כאובייקט מסווג `ElementRef`, עם אובייקט זה נוכל לבצע כל שינוי שנרצה.

אובייקט זה ישנו המאפיין `nativeElement` שהוא האלמנט ב DOM. מאפיין זה מכיל את אותו אובייקט לו היינו מבקשים באמצעות `document.getElementById(id)` מה `directive` directive שאנו יכולים לבצע עליו כל קוד JS שאנו מכירים בהקשר לאלמנטים.

בדוגמה להלן נקבע את צבע הרקע של האלמנט:

```
constructor(el: ElementRef) {
  el.nativeElement.style.backgroundColor = 'yellow';
}
```

דוגמה לימוש `directive`:

```
<p appHighlight>Highlight me!</p>
```

מאזינים לאירועים על האלמנט ב DOM

ניתן להגדיר מאזינים לאירועים העוברים על האלמנט שמאරח את ה directive ולבצע שינויים באלמנט בהתאם לכך.

לצורך כך נגדיר מתודות המבצעות פעולה רצiosa כלשי כתגובה לאירוע, ו"נקשט" אותו ב decorator `HostListener`. `HostListener` צריך לקבל את שם האירוע אליו עליו להאזין.

דוגמה:

Angular v.7

```

@HostListener('mouseenter')
onMouseEnter() {
    this.highlight('yellow');
}

@HostListener('mouseleave')
onMouseLeave() {
    this.highlight(null);
}

private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
}

```

בדוגמה זו האזנו לאירועי כניסה ויציאה עבור מהאלמנט המארח, וצבענו את הרקע בהתאם לזהות.

שליחת פרמטרים ל directive

ניתן לשולח פרמטרים ל directive בדומה لما שהכרנו עד היום כ. באונה הדרך בה העברנו נתונים לקומפוננטה – כך נעביר נתונים ל directive.

נדיר את המאפיינים אותם נרצה לקבל מבחן כ(`@Input()`) ווכל לעשות בהם שימוש.

לדוגמה:

במחלקה של directive נaddir מאפיין `highlightColor`:

```
@Input() highlightColor: string;
```

באלמנט העושה שימוש ב directive זה נשלח את הצבע הרצוי לנו:

```
<p appHighlight [highlightColor]="'orange'">Highlighted in orange</p>
```

במידה ונaddir alias ל שיהיה כשם הסלקטור של directive נוכל לחסוך בימוש `highlight` ולשלוח בבת אחת עם הגדרת directive את הערך הרצוי.

נעשה זאת כך:

```
@Input('appHighlight') highlightColor: string;
```

נமמש זאת כך:

```
<p [appHighlight]="color">Highlight me!</p>
```

חשוב! יש לזכור להוסיף את directive למודול אליו הוא שיופיע במערך ה declarations.

Pipes

Pipes הם בעצם כינויו של מנגנון אחד נבניר את המידע המקורי איך שהוא, ומצידם השני המידע יצא שונה. כל זה מתייחס לתצוגת המידע בלבד. המידע עצמו נשאיר כמו שהוא.

Pipes מבניות

באנגולר קיימים pipes מובנים ושימושיים לדוגמה:

`:date`: הצגת תאריך בפורמטים שונים

`:lowercase / uppercase`: הצגת מחרוזת באותיות גדולות או קטנות

`:percent`: הצגת מידע באחוזים

`:currency`: הצגת סוג מטבעות

`:decimal`: מציג מספר בתבנית דצימלית. ניתן להגדיר כמה ספרות מינימום לפני הנקודה, כמה ספרות מינימום לאחר הנקודה וכמה ספרות מקסימום יוצגו לאחר הנקודה.

`:slice`: עברו מערך, מציג ערכים מאינדקס מתחילה נתון ועד אינדקס יעד נתון (לא כולל). אינדקסים מתחילה מapse כרגע.

צורת השימוש בסופק היא " `כתיבת הערך בתוספת התו | ולאחריו שם הסופק`:

```
@Component({
  selector: 'hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

פרמטרים

ניתן לשולח פרמטרים לpipe אם יש צורך.

כמו למשל בדוגמה זו, בה ניתן פרמטר באיזה סוג פורמט להציג את התאריך:

```
<p>The hero's birthday is {{ birthday | date:'MM/dd/yyyy' }} </p>
```

שידור pipes

ניתן לשלב כמה pipes ביחד, כמו למשל:

```
 {{ birthday | date | uppercase}}
```

Custom pipes

ניתן בקלות רבה ליצור pipes מיוחד לבך. יש ליצור מחלקה המממשת את PipeTransform וע"י המתודה transform להחזיר את הערך בצורה הרצiosa. המחלקה תקבל @Pipe ותגדיר מה השם של ה pipe לשימוש. להלן דוגמת קוד לימוש pipe המתקבל ערך מספרי כלשהו וכפרמטר מקבל ערך מספרי נוסף ומוחזק את הערך הראשון בחזקת הערך השני.

```
import { Pipe, PipeTransform } from "@angular/core"
@Pipe({
  name: "pow"
})
export class PowPipe implements PipeTransform {
  transform(value: number, exponent: string) {
    var exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

צורת השימוש בסידוק זה תהיה כה:

```
<p>{{5 | pow: "2" }}</p>
```

הפונקציה transform מקבלת בכל מקרה את הפרמטר value שהוא המידע עליו ה pipe עובד. שאר הפרמטרים אלו פרמטרים שניית להוסיף במידה ויש צורך. חשובו יש לזכור להוסיף את ה pipe למודול אליו הוא שייך במערכת declarations.

Lazy Loading

כאשר מתוכנים אפליקציה – מחלקים אותה למודולים, בד"כ יהיה מודול שהוא **shared** – מודול משותף עם שירותים וקומponeנטות ומרכיבים נוספים שימושתיים ויבאו לידי שימוש ברוב אזור האפליקציה.

לעתים, יהיו מודולים אשר לא בהכרח שיבאו לידי שימוש כר שחייב לטען את כל הקוד שלהם בעליית המערכת, אלא היה טוב אם ניתן היה לטען אותם רק במקרה וינסו לגשת לתוכנים של אותו מודול.

התנהגות זו נקראת **loading lazy** – טינה עצלה. טינה שאינה מופעלת מראש, אלא רק אם יש צורך.

ניתן למשוך **loading lazy** באופן גלובלי בצורה פשוטה ביותר. ישנן שתי אפשרויות לטינה דינמית: טינה ע"י **routing** וטינה ע"י **קוד**.

טינה ע"י **routing**

appModule מגדיר לעצמו את ה **RouterModule** עם כל הניווטים הקיימים במערכת. אולם, כאשר היו מספר ניווטים הקשורים כלום למודול מסוים במערכת, נoon יהיה ליצור **routing** נפרד למודול זהה ולנהל בו את כל הניווטים שבאחריות המודול הספציפי.

כך שאם נכתבו ונתכון נכון המערכת – כל מודול יהיה אחראי על הניווטים שלו, וב **appModule** נדרש רק לתת הפניה ראשית לכל מודול. בכל מודול יוגדרו כבר כל תתי הניווטים שלו. בצורה זו קל מאד למשוך **loading lazy**.

ב **appModule** אנו ניתן הפניה ראשית במערך הניווטים, הפניה זו תיגש לקוד בו המודול מוגדר, הקוד הזה יטען וימשייר את העבודה ממנו והלאה.

כך בעצם לא נדרש לטען מראש ב **appModule** את כל הקוד של תתי המודולים, אנחנו נפנה במידה ויתבצע ניווט תואם – למודול המתאים, הקוד של המודול יטען והניווט ימשיך להלאה לתוך המודול פנימה.

ניתן כאן דוגמה:

נתחיל עם התת מודול, נדגים זאת על מודול **account** שמייצג את כל אפשרות החשבון של משתמש באתר. מודול זה לא בהכרח שיבוא לידי שימוש בכל כניסה לאתר ولكن נרצה לטען אותו ב **lazy**. אם נريץ את הפקודה:

`ng g m account –routing`

ישוצר לנו מודול בשם **AccountModule** ומודול נוסף **AccountRoutingModule** שנמצא בתוך ה **AccountModule** של imports.

AccountModule יראה כך:

Angular v.7

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { AccountRoutingModule } from './account-routing.module';

@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    AccountRoutingModule
  ]
})
export class AccountModule { }
```

יראה כך: AccountRoutingModule

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class AccountRoutingModule { }
```

נשים לב ל `RouterModule.forChild(routes)`, זאת מכיוון שאנחנו מגדירים כאן את הRoutes בתוך מודול ולא למודול הראשי.

לאחר שייצרנו את תת המודול, ניתן הפניה ב `appModule` למודול זה:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { ... },
  { path: "account", loadChildren: "./account/account.module#AccountModule" },
  { ... }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

ההפניה זו אומرتה כך: כאשר יש לך ניוט账户 לזה נמצא בעצם בתת מודול – loadChildren – והוא מטען אותו ואת ילדיו, מאיפה לטעון אותו? מההמחרוזת אותה רשםנו ב – loadChildren – מחרוזת זו היא הכתובת לקובץ של המודול בתוספת # ושם המחלקה של המודול.

Angular v.7

אם נריץ את האפליקציה כעת, נוכל לראות איך זה עובד. בתחילת הכתובת `/http://localhost:4200` נראה את `AppComponent` לפני מה שיש בתוכה.
אם נפתח את ה `tools developer` (F12) ונראה מה קורה ב `network` נראה שקובצי הקוד שיידן מראש הם:

The screenshot shows the Network tab in the Chrome Developer Tools. At the top, there are tabs for Elements, Console, Sources, Network, and Performance. The Network tab is selected. Below the tabs are buttons for Group by frame, View, and a filter input. Underneath is a table with columns for Name, Size, and Duration. The table contains five rows corresponding to the files listed in the code snippet above: runtime.js, polyfills.js, styles.js, vendor.js, and main.js. The total duration for all files is 5ms.

Name	Size	Duration
runtime.js		100 ms
polyfills.js		200 ms
styles.js		300 ms
vendor.js		400 ms
main.js		5

אם ננווט כעת בשורת הכתובת ל `http://localhost:4200/account` או כל ניוט כלשהו, נוכל לראות ב `network` איך יורד קובץ חדש ומוסך בשם `account-account-module.js`.

The screenshot shows the Network tab in the Chrome Developer Tools after navigating to `http://localhost:4200/account`. The interface is identical to the previous screenshot, but the table now includes an additional row for the file `account-account-module.js`, which has a duration of 400 ms. The other files (runtime.js, polyfills.js, styles.js, vendor.js, and main.js) still appear in the list.

Name	Size	Duration
runtime.js		100 ms
polyfills.js		200 ms
styles.js		300 ms
vendor.js		400 ms
main.js		400 ms
account-account-module.js		400 ms

Angular v.7

קובץ זה מכיל את הקוד של accountModule והוא לא ירד בתחילת אלא רק לאחר שינויו למודול זהה.

כמובן שהדוגמה כאן היא בסיסית ביותר, ללא קומפוננטות וכל מה שמודול אמיתי מכיל, אולם זה הרעיון בבסיסו.

טעינה ע"י קוד

ניתן לטעון מודולים שלא ע"י routing, אלא ע"י קוד.

כאשר יש צורך לטעון קומפוננטה ממודול ואין אפשרות לטעון זאת ע"י ניווט,opsisות שומות, כמו למשל שהקומפוננטה אמורה להשתלב כחלק ממשך אשר הניות שלו אינם בידים שלה. לצורך כך ניתנה אפשרות לטעון קוד באופן דינמי.

עשה זאת באמצעות פונקציית import המקבלת את הנטייב לקובץ של המודול (נטיב מודיק לפ' מיקומו ביחס לקומפוננטה הטוענת), נבצע קומpileציה על הקוד ונקבל לדינם את כל יוצריה הקומפוננטות שבמודול זהה.

נבחר את יוצר הקומפוננטה הרצוי וニיצר דרכו את הקומפוננטה בתוך האלמנט המועד.

הקוד יראה כך:

```
import { Component, ComponentFactory, OnInit, Compiler,
ModuleWithComponentFactories, ViewChild, ViewContainerRef } from '@angular/core';

@Component({
  selector: 'home-page',
  template: `<div #container></div><button (click)="openModalLogin()"></button>`
})
export class HomePageComponent implements OnInit {
  @ViewChild('container', {read: ViewContainerRef}) private _container:
ViewContainerRef;

  constructor(private _compiler: Compiler) {}

  ngOnInit() {}

  async openModalLogin() {
    const module = await import('../modals/modals.module');
    const moduleWithComponents: ModuleWithComponentFactories<any> = await
this._compiler.compileModuleAndAllComponentsAsync(module.ModalsModule);
    const componentFactories = moduleWithComponents.componentFactories;
```

Angular v.7

```
const appLoginFactory: ComponentFactory<any> =  
componentFactories.find((factory: ComponentFactory<any>) => {  
    return factory.selector === 'app-login';  
});  
this._container.createComponent(appLoginFactory);  
}  
}
```

Change Detection

Change detection זהו המנגנון של אングולר הלודכ' כל שינוי שמתבצע במערכת שעשו להשפיע על התצוגה. בעבר כל שינוי זהה – אングולר מרדנזר את כל הקומפוננטות מקומפוננטת השורש ועד לאחרן היצאים.

המנגנון הזה בניי על ספריית `zone` זו היא ספרייה המזענדת ליצור עין חדרים לקוד. כמו שבכל חדר – ישנה דלת וכל יציאה מן החדר דורשת מעבר בדלת. כך בעצם הספרייה הזו תוחמת בחדר אחד כביכול אוסף של קודים, וכל פעולה של קוד שרצאה לצאת החוצה ולבצע משהו ישנה דלת כלשהיא שהוא צריך לעבור בה.

כך בעצם אングולר תוחם את כל הפעולות שלו בחדר אחד כביכול – וכל יציאה מן החדר – אングולרחושד שהיא עשויה להשפיע על התצוגה ולכן מפעיל change detection הגורם לרינדור כל הקומפוננטות.

סוגי הקודים שבאופן אוטומטי יגרמו לאングולר ללקוד אותם ולבצע change detection יהיו לדוגמה:

- `setInterval` `setTimeout` – timer
- `Promise` – קוד אסינכרוני שישנה המתנה לגובהתו
- `Observable` – `data-stream` ציר מידע פתוח שישנה המתנה לפולס' מידע ממנו.
- `http` – קריאות החוצה לשרת
- `events` – אירועים

מנגנון זה הינו מנגנון מעולה אולם יש בו חיסרונות ממד גדול – עבור כל קוד מן הסוגים שנזכרו לעיל – כל המערכת יכולה תתרנדר מחדש גם אם בסופו של דבר הקוד לא גרם לפחות שינוי בתצוגה או שהוא גרם לשינוי במקומות מסוימים ממד אבל לא היה צריך בשבייל זה לצירר מחדש את כל הקומפוננטות מההתחלה ועד הסוף.

אם נחשוב על בעיות הביצועים שיכולים להיגרם מכך נגיע למסקנה שהמנגנון הזה חייב להיות נתון לשליתה.

אנגולר אכן מאפשר להגדיר עבור כל קומפוננטה באיזו אסטרטגיה היא פועלת, האם באופן ברירת המחדל כפי המתואר לעלה, או באופן שנקרא `OnPush` – אופן זה אומר שרק כאשר "נדחפים" ערכים חדשים לקומפוננטה רק אז יבוצע מנגנון ה `change detection`.

צורת הגדרת הקומפוננטה תיראה כך:

```
import { Component, OnInit, ChangeDetectionStrategy } from '@angular/core';

@Component({
  selector: 'app-user-profile',
  templateUrl: './user-profile.component.html',
  styleUrls: ['./user-profile.component.scss'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class UserProfileComponent implements OnInit {
```

Angular v.7

```
constructor() { }

ngOnInit() {
}

}
```

הפעולות שיגרמו לקומפוננטה המוגדרת כ OnPush להתרנדר מחדש יהיו:

- Input – ערכים חדשים שייכנסו למשתנים המוגדרים כ ()@Input

- events – כל האירועים שיקרו, כולל output

- change – המתנה לנתחים ע"י async pipe subscribe) בקוד לא יגרום ל change detection

- כאשר באופן יוזם מבצעים קרייה להפעלת detection ע"י הזרקת ChangeDetectorRef

```
constructor(private cdr: ChangeDetectorRef)
```

ומפעילים אותו כך:

```
this.cdr.detectChanges()
this.cdr.markForCheck()
```

כאשר יבוצע change detection במערכת בגל קומפוננטה אחרת – הקומפוננטות המוגדרות כ onPush לא יצטירו מחדש, אנגולר ידלג עליהם וישאיר אותן כמוות שהן.

מומלץ מאד לכתוב את כל הקומפוננטות כ OnPush – ובמידה וישן בעיות לנסוטות לפטור אותן, ורק במקרה ולא ניתן לפטור אותן – רק אז להגדירן שיעבדו כברירת המחדל.

הידושים ועדכוניים מאנגולר 6 ו 7

CLI חדש, עם פקודות חדשות

קובץ ה `angular.json` הוחלף ב `angular-cli.json`

`ng add`

פקודה זו דומה ל `npm install` או ל `npm` היא מסוגלת להווסף אף קבצים חדשים ולעדכן קבצים קיימים בתחום הקוד עצמו, ולא רק בתיקיית `node_modules`, בהקשר לחבילה אותה ביקשנו להווסף.

`ng update`

פקודה זו גורמת לעדכון חביבה כולל ההשלכות הנילוות.

`providedIn: 'root'`

נוספה אפשרות להגדרת המחלקה הצורכת את השירות מתוך השירות עצמו. עד אנגולר 6 השירותים הוגדרו רק מתוך המחלקה הצורכת.עתה ניתן להגדיר בשירות עצמו – לאיפה להזrik אותו.

Angular elements

ניתן לייצר מוקומפוננטות של אנגולר אלמנטים לשימוש כאלמנטים המכילים `html` וקוד `js` בלבד. יש להזה יתרון גדול בהוצאת קומפוננטות לשימוש בסביבות שאין אנגולריות.

Rxjs 6

אנגולר בגרסה זו עבר לעבד עם `6 rxjs`. בגרסה זו בוצעו מספר שינויים במיקומי האופרטורים ואופן השימוש בהם.

PWA

יכולת חדשה המאפשרת לאפליקציה אנגולר `web` להתנהג אפליקציית `native` ללא צורך בהתקנה. יש אפשרות לעבוד `offline`, להרשם ולשלוח `notification`.

Schematics

האפשרות לייצר פקודות ריצה נקראת `schematics`. ניתן לייצר פקודות בדומה לפקודות `ng generate` וכן, הפקודות הללו יכולות לייצר קבצים חדשים, לעדכן קוד בקבצים קיימים, להוריד חבילות, לעשות ממש הכל.

Workspace

ניתן לעבוד כ `solution` עם מספר פרויקטים בפרויקט אחד – `workspace`. היתרונות של זה הוא בתיקיית `node_modules` אחת משותפת לכל הפרויקטים.

Library

ניתן לייצר פרויקט שהוא אינו אפליקציה, אלא ספריה שייצרנו אותה.

Angular v.7

Angular universal

.server side rendering – SSR
נקרא גם

האפשרות לрендר בצד שרת את הקומפוננטות ובכך לקצר זמן טעינה.

Angular 7

נוספו שיפורים ביצועים, drag and drop וvirtual scrolling ועוד תוספות מינוריות. השינויים המשמעותיים היו בעברanganular 6.

stop

slice

when let -> we make things easier, better for var - var : TypeScript
- missing with local scope & pr

let user : {
 \downarrow
 string/number/boolean | any -

- const

function slice (array: any): string[]
function slice (array: any, start?: number, end?: number,
 params: any, ...rest: any)

(foreach, etc) for user
var i in user
of value
getMessage() - get user
return this.message;
}

setMessage(text: string) - set user

constructor

- which be ctor if class extends - etc
super()

instanceof - is user

slice

Node - v

npm install -> @angular/cli

new angular

file main -> main
ng serve - use browser

export-class

create w/ (ctrl + p) new file
import -> new

generate ->

, /src /main - []

src

cd src/main - (ctrl + o)

ng serve -o app (ctrl + l)

<app> "M"

Angular has modules

- files with .ts, .js & .css, .html etc are in /src/app-root
- files like <app-root>/app-root

(app -> component -> service -> model -> class)
-> if needed angular has lots of them we can use

byndist - ref

{<title>} if

*ngFor - for map

*ngIf - if else

html ts will be converted into js

Angular 3 - Ng c

most files - output

will return no proper value except EJS / angular files - Input

local reference

<input type="text" #a>

<input type="text" #b>

then ref value same at both - (click) = print(a, b)

Event Emitter, Output - imports module - node modules
require('node').emit

SPA - signal page Application

url / & id

subscribe -
params, " url -> function person
params, " url -> function person
params, " url -> function person

03 0535991

pipes

we list all the pipe from core module we could use pipe in re-usable

useful means you can use it again &

lowercase in /

in uppercase

ng g pipe pipe -a,3, you

pipe (1 -a,3, you, 3)

pipeTransform interface

in type script (2)

you pipe { import { } from 'app.module'