

The background of the slide is a blurred image of a financial market display. It features several stock market indices and their corresponding values. Visible text includes "OMX COPENHAGEN 25 INDEX" with a value of 1172.94, "OMX RIGA GI" with a value of 984.13, "OMX18" with a value of 27956.04, and "OMX18" with a value of 28289.06. There are also "Buy" and "Sell" indicators. The overall color scheme is dark with red and green highlights for price changes.

Angular & Web API (C#)

דור זילכה

ברוכים הבאים

הצגת המרצה
והכירות עם
הקורס

הכירות עם
המפתחים
בקורס

TYPESCRIPT

JavaScript

- number
- string
- boolean
- null
- undefined
- object

TypeScript

- any
- unknown
- never
- enum
- tuple

Why TS

TS differences from JS

TS Transpiler

Why angular choose TS

var vs let vs const

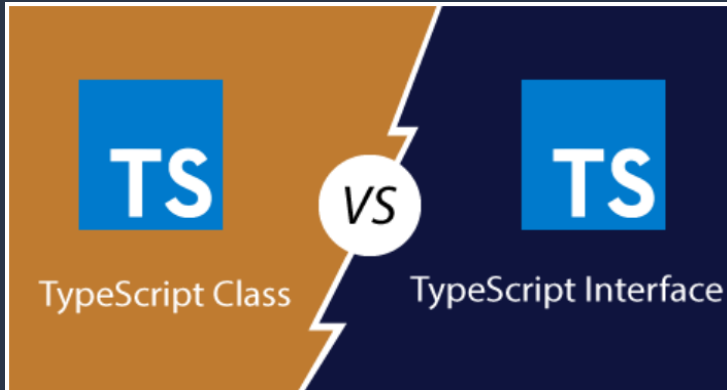
- const – never changes
- let bound to its scope {}
- var – unbound
- Side note on components
Because your features (component, service, pipe, etc) are classes.
In classes, you define properties with accessors, not scoped variables.

```
function run() {  
  var foo = "Foo";  
  let bar = "Bar";  
  
  console.log(foo, bar); // Foo Bar  
  
  {  
    var moo = "Mooo"  
    let baz = "Bazz";  
    console.log(moo, baz); // Mooo Bazz  
  }  
  
  console.log(moo); // Mooo  
  console.log(baz); // ReferenceError  
}  
  
run( );
```

First usage of TSC

- `npm install -g typescript`
- `tsc -init`
- Introduce `tsconfig.json`
 - Source map
 - Out dir
 - Source/root
 - lib
- Write code and let's see transpiler at work
- Debug out code – `lunch.json` : `prelunchtask` : `"tsc: build - tsconfig.json"`

Interface, Types and Inheritance



A class cannot disappear during the compilation of code.

The member of a class can be public, protected, or private.

Interface completely disappeared during the compilation of code.

The members of an interface are always public.

```
class Person{  
}  
  
class Worker extends Person{  
}  
      ↑          ↑  
    נגזרת   בסיס
```

The methods of a class are used to perform a specific action.

The methods in an interface are purely abstract (the only declaration, not have a body).

Let's code an example

Import & Export

Let's code example

Task time

- Install typescript globally : `npm install -g typescript`
- Create folder and start a new ts project : `tsc --init`
- Set the `tsconfig.json` so that the project code will be under `src` folder
- Set the `tsconfig.json` so that the transpiled JS file will be under `dist` folder
- Set the `tsconfig.json` so you will have a `sourceMap`
- Set the `tsconfig.json` so you will have a `lib` configured
- Write policeman script like shown
 - Add waiting time after weaning shoots async timeout using `setTimeout`
`import {setTimeout} from "timers/promises";`
Hint : use `Promise` and see `js` file after trasnpile

SPA

1.1. הצורך בתשתיות SPA

SPA – Single-Page Application.

יישומי דף יחיד הם יישומי רשת, שמטרתם לתת חוויית משתמש מהירה וזורמת יותר, הדומה לתוכנת מחשב רגילה (שאינה יישום רשת). ביישומים אלו, כל הקוד הדרוש (HTML, CSS, JS) מגיע לדפדפן בטעינת דף אחת, ומשאבים נוספים נטענים באופן דינמי, בדרך כלל, כתגובה לפעולות המשתמש. דף האינטרנט לא מבצע טעינה מחדש, אולם כתובת האינטרנט עשויה להשתנות מעט, על מנת לתת למשתמש הבנה יותר טובה של הניווט בדף.

SPA הוא ההתפתחות הטכנולוגית של multi-page application ושימוש ב-AJAX.

1.2. יתרונות SPA

- זמן טעינה מהיר יותר של דף.
- טעינה ברקע בצד השרת, מגדילה את חוויית המשתמש.
- אין צורך לרענן דפים בשרת בכל פעולת משתמש.
- אין תלות חזקה בין צד השרת לצד הלקוח.
- הפיתוח תואם לעולם אפליקציות המובייל, משום שניתן להשתמש באותו שרת לאפליקציית מובייל שונה (מסך קטן יותר, וכו').

Angular

- שפת Angular - מודרנית ובעלת יכולות פיתוח גבוהות יותר, בשילוב מובנה עם type script. זו שפה מודולרית וקלה ללמידה (בהשוואה ל Angular 1.x).
- ארכיטקטורת Angular - הינה מודולרית עם תלות פחותה בין האובייקטים (בהשוואה ל- Angular 1.x). יכולת זו מובילה לפיתוח נכון ארכיטקטוני, שחוסך תקלות.
- ביצועים ב- Angular - טובים יותר. היא הופכת תבניות לקוד. תבניות אלו עוברות אופטימיזציה, המשפרת (מבחינת ביצועים) משמעותית את קוד ה-Java script המתקבל.

Components

"We're not designing pages, we're designing systems of components."

- Stephen Hay -

המרכיבים הבסיסיים של components:

1. **Encapsulation – הכמסה** – Encapsulation פירושו שילוב לוגיקה ו-data יחד בתוך container בודד. כאשר encapsulation נכון מסייע לנו בארגון הקוד שלנו. ללא צורך לזכור ולהבין לעומק את כל ה-internal logic של המכולה הסגורה ואת לוגיקת הוואלידציות המורכבת שיצרנו עבור הנתונים. ובכך לעבוד ברמה גבוהה יותר הפשטה (higher-abstraction level).

ב-components השימוש ב-Encapsulation מכוון ליצירת רכיבים קטנים ותמציתיים, שבסופו של דבר ישולבו למערכת של רכיבים. ובכך מתאפשרת במהלך הפיתוח, היכולת להתמקד בתוכן ובלוגיקה הפנימית של רכיב אחד בלבד, ואספקת גישה לקוד ספציפי בקלות, תוך כדי יצירת האפשרות להתמקד בשכבה אחת של הקוד, ולבטוח ב-underlying implementations הכמוסים בתוך הרכיבים.

2. **Composition - הכלה** – כל component הוא רכיב הוא עצמאי, אבל יכול לתקשר עם רכיבים אחרים וליצור רכיב גדול יותר על ידי הכלה פנימית של רכיבים אחרים.

Components

Components .2.3.2

ה – Component מאפשר לנו לייצר תגיות חדשות בשפת HTML. לכל תגית שנייצר נוכל להגדיר מה הלוגיקה העסקית שלה (typescript), מה עימוד הרכיבים בתוכה (HTML) ומה העיצוב שלה (CSS).

```
/** Created by shai vashdi on 27/08/2016. ...*/  
import {Component} from '@angular/core';  
  
@Component({  
  selector: 'headerControl',  
  templateUrl: './header.control.template.html',  
  styleUrls: ['./header.control.style.css'],  
})  
  
export class HeaderComponent {  
}
```

Directives

היכולת להוסיף Html Attributes חדשים לשפה.

Directive מתפקד כעלוקה על Element קיים, לכן אין לו HTML ואין לו CSS אלא אך ורק ts file שזה אומר אך ורק קוד.

דוגמאות לDirectives שניתן ליצור :

- Attribute – Only-hebrew שכאשר נשים אותו על אלמנט מסוים הוא לא יאפשר להזין בו שום דבר חוץ מעברית.
- Attribute – Deault-image שנוסיף לתגית image שבו נוכל להגדיר אם התמונה לא נמצאה איזו תמונה לשים.

ישנם שני סוגים של Directive:

- directive – Structural directive זה משנה את ה-DOM על ידי הוספה והסרה של אלמנטים, למשל: ngIf ו-ngFor.
- Attribute directive – משנה את המראה או ההתנהגות של אלמנט, למשל: NgStyle.

Services

אפליקציות מורכבות ממספר תת מערכות, למשל: logging, data access, caching.

המושג Service ב-Angular, הינו כימוס (encapsulation) של פונקציונאליות כלשהי, כדי שתוכל לספק את הפונקציונאליות הזו, באופן בלתי תלוי, לשאר חלקי האפליקציה.

ב-Angular, ה-services מועברים דרך ה-constructor באמצעות dependency injection.

מבחינתנו, כל מחלקה שמכילה פונקציונאליות היא Service. מחלקה שמכילה רק שדות נתייחס אליה כModel או Entity.

Dependency injection

הזרקת תלויות (באנגלית: dependency injection), הינה תבנית עיצוב, המאפשרת בחירה של רכיבי תוכנה, בזמן ריצה (ולא בזמן ההידור). תבנית זו יכולה, לדוגמה, לשמש כדרך פשוטה לטעינה דינאמית של plug-ins או בחירה באובייקטי דמה (mock objects) בסביבות בדיקה, במקום להשתמש באובייקטים אמיתיים של סביבת הייצור. תבנית עיצוב זו, מזריקה את האלמנט שתלויים בו (אובייקט או ערך, וכדומה) אל היעד שלו, בהתבסס על ידיעה של צרכי היעד.

המטרה העיקרית היא למנוע צמידות בין מחלקה אחת למחלקה אחרת.

מנגנון ה-dependency injection ב-Angular, מורכב משלושה חלקים עיקריים:

- Injector – הרכיב שמייצר את ה-instance של האובייקט המוזרק.
- Provider – הרכיב שמגדיר כיצד לייצר את האובייקט. למעשה, ממפה את הדרישה לייצור האובייקט אל פונקציית factory, המייצרת אותו בפועל.
- Dependency – סוג האובייקט אותו מבקשים לייצר בדרך זו.

NgModule

זהו רכיב המסייע לסדר ולארגן את האפליקציה. התשתית מחליטה כיצד להדר (compile) ולהריץ את הקוד, על סמך ה-metadata שניתן לאובייקט.

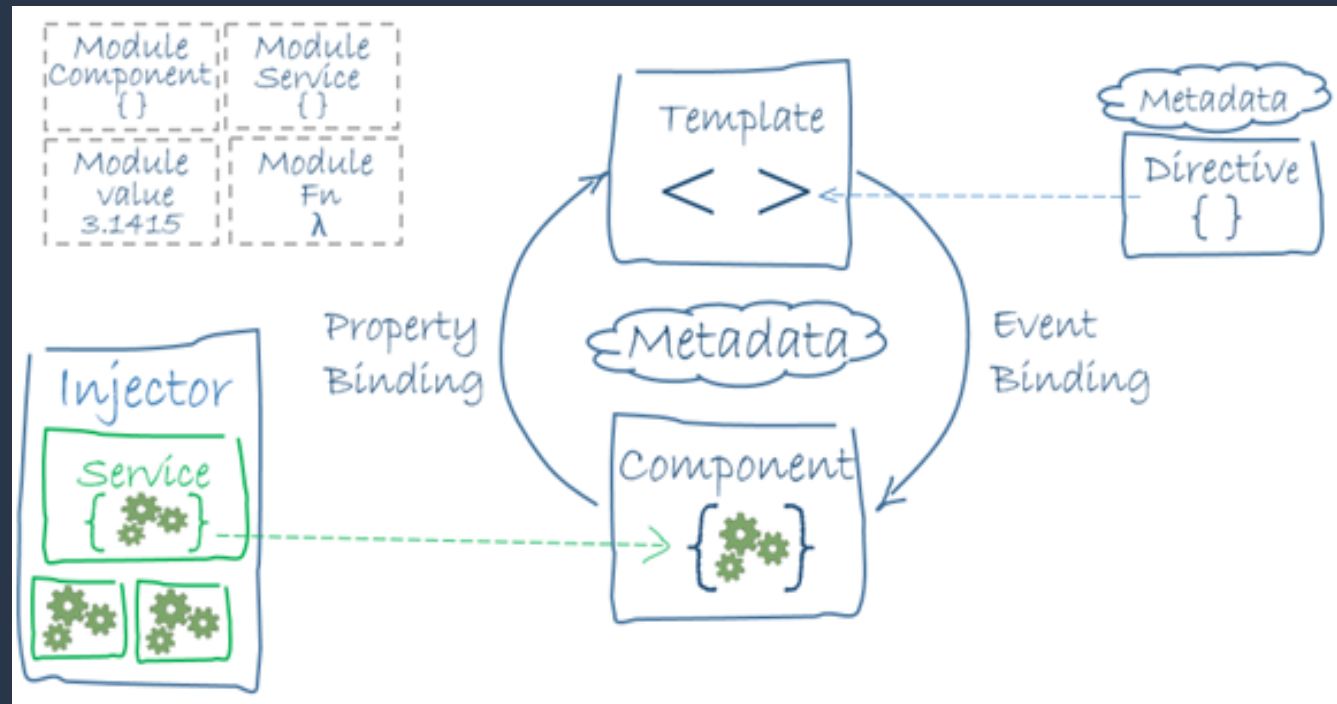
מודולים נחלק על בסיס מספר עקרונות.

1. הפרדת תשתית מאפליקציה. כמו שבסביבות אחרות היינו מייצרים DLLים עם רכיבים לשימוש חוזר, גם כאן נייצר מודולים תשתיתיים שנוכל לעשות בהם שימוש חוזר. בין לוגיקה שחוזרת על עצמה מצד אחד לרכיבי UI שחוזרים על עצמם מצד שני.

2. חלוקה על בסיס נושאים – הרבה מערכות מכילות מספר תתי מערכות. לדוגמה בכל מערכת יש את המערכת עצמה ויש את איזור הניהול שלה. גם כאן נגדיר את המערכת עצמה ואת מערכת הניהול כשני מודולים נפרדים. מודולים שמוגדרים בצורה נפרדת יסייעו לנו לעליה ראשונית מהירה יותר של המערכת ותיתן לנו את האפשרות לטעון מודול רק כאשר באמת עושים בהם שימוש.

Angular Hello world

- `npm install -g @angular/cli`
- `ng new my-app`
- `node modules`
- `packages.json`
- `npm start`
- `ng g c first-comp`
 - `encapsulation: ViewEncapsulation.*`



Lifecycle hooks

- A component instance has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views. The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed. The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM. Directives have a similar lifecycle, as Angular creates, updates, and destroys instances in the course of execution.
- Your application can use [lifecycle hook methods](#) to tap into key events in the lifecycle of a component or directive to initialize new instances, initiate change detection when needed, respond to updates during change detection, and clean up before deletion of instances.

Component & Directives : Life cycle

כעת נתייחס לתהליכים השונים בחייו של הרכיב, ולאופן בו נוכל לכתוב לוגיקה נוספת, בכל שלב בחייו של הרכיב. זהו שלב ה- LifeCycle hooks.

נתבונן בטבלה:

שלב	מטרה
<u>ngOnInit</u>	זהו שלב <u>האיתחול</u> של הרכיב או ה-directive, לאחר שהוכנסו ה-inputים שהוגדרו לו.
<u>ngOnChanges</u>	זהו השלב שבו הרכיב מקבל את ה-inputים שהוגדרו לו או כאשר הם משתנים.
<u>ngDoCheck</u>	זהו שלב בו ניתן לבדוק שינויים שהתשתית לא הבחינה בהם בעצמה. הוא נקרא בכל פעם שהתשתית מחפשת שינויים ברכיב ומסייע לה להרחיב את הבדיקה.
<u>ngOnDestroy</u>	זהו שלב הריסת אובייקט הרכיב או ה-directive. בשלב זה ננקה רישום ל-event-ים או רישומים ל-observables, על מנת למנוע דליפת זיכרון.

More Component : Life cycle

בשונה מ-directive, ל-component שלבים נוספים:

שלב	מטרה
<u>ngAfterContentInit</u>	זהו השלב שבו התשתית מכניסה את רכיבי הבנים של התצוגה שלו.
<u>ngAfterContentChecked</u>	זהו השלב שבו התשתית בודקת את ה- bindings של רכיבי הבנים של הרכיב.
<u>ngAfterViewInit</u>	זהו השלב שבו התשתית יוצרת את התצוגה של הרכיב.
<u>ngAfterViewChecked</u>	זהו השלב שבו התשתית בודקת את ה- bindings של התצוגה של הרכיב.

When

נתבונן על תזמון האירועים בחייו של רכיב או directive:

שלב	מתי?
<u>ngOnChanges</u>	לפני <u>ngOnInit</u> ובכל פעם שמתבצע שינוי בקלטים של הרכיב.
<u>ngOnInit</u>	לאחר הפעם הראשונה ש- <u>ngOnChanges</u> נקרא.
<u>ngDoCheck</u>	בכל פעם שהתשתית מחפשת שינויים ברכיב.
<u>ngAfterContentInit</u>	לאחר שהתשתית מכניסה את רכיבי הבנים של הרכיב לתצוגה שלו.
<u>ngAfterContentChecked</u>	לאחר שהתשתית בודקת את ה-bindings של רכיבי הבנים של הרכיב.
<u>ngAfterViewInit</u>	לאחר שהתשתית יוצרת את התצוגה של הרכיב.
<u>ngAfterViewChecked</u>	לאחר שהתשתית בודקת את ה-bindings של התצוגה של הרכיב.
<u>ngOnDestroy</u>	בדיוק לפני שאנגולר הורסת את הרכיב או ה-directive.

Difference between One-way Binding and Two-way Binding

- **One way binding:**
- In one-way binding, the data flow is one-directional.
- This means that the flow of code is from typescript file to Html file.
- In order to achieve a one-way binding, we used the property binding concept in Angular.
- In property binding, we encapsulate the variable in Html with square brackets([]).
- We will understand this concept through an example in order to make it more comprehensible.

```
<h3>Displaying the content without one way binding</h3>

<hr />

<h3 [textContent]="title"></h3>
```

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

@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"],
})
export class AppComponent {
  title = "Displaying the content with one way binding";
}
```




Difference between One-way Binding and Two-way Binding

- **Two-way binding:**
- In a two-way binding, the data flow is bi-directional.
- This means that the flow of code is from ts file to Html file as well as from Html file to ts file.
- In order to achieve a two-way binding, we will use ngModel or banana in a box syntax.
- To make sure the app doesn't break, we need to import 'FormsModule' from '@angular/forms.
- Any changes to the view are propagated to the component class. Also, any changes to the properties in the component class are reflected in the view.
- To bind two properties in order to two-way binding works, declare the ngModel directive and set it equal to the name of the property.

```
<input [(ngModel)]="data" type="text">  
  
<hr>  
  
<h3> Entered data is {{data}}</h3>
```

```
@Component({  
  selector: "my-app",  
  templateUrl: "./app.component.html",  
})  
export class AppComponent {  
  data = "Ram and Syam";  
}
```

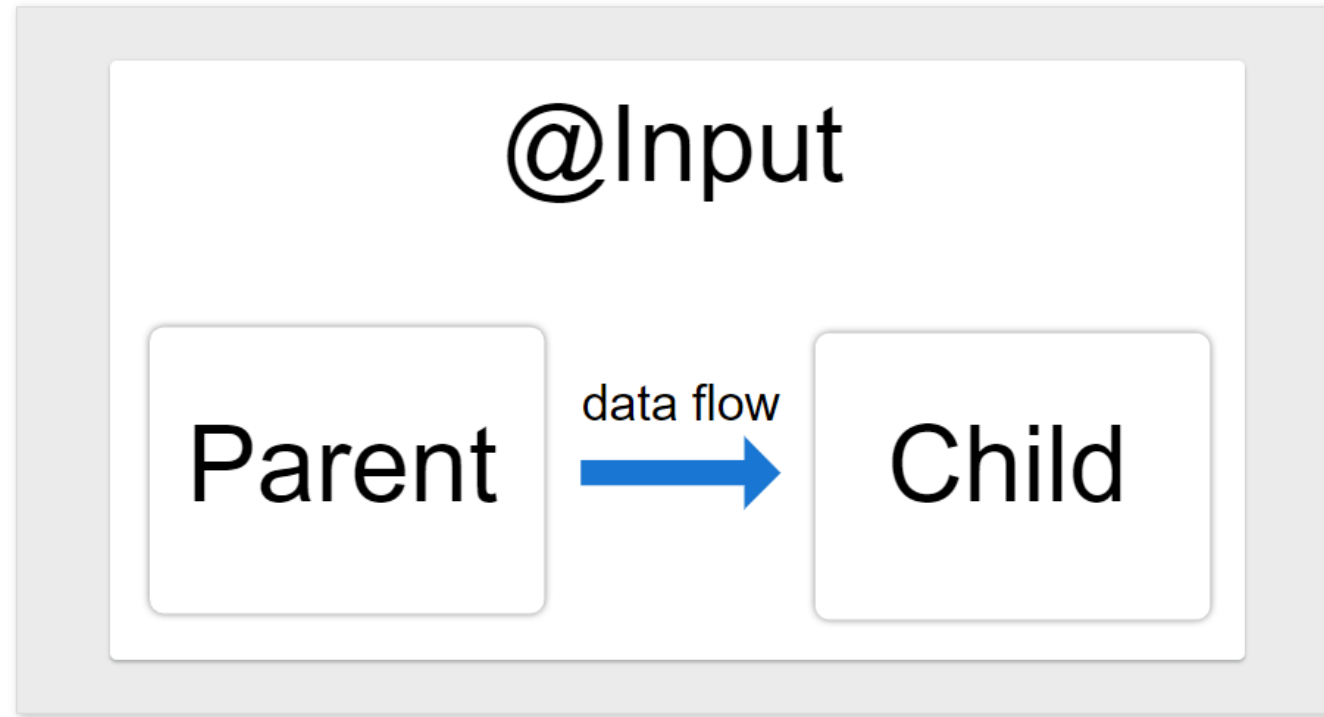
Syntax

TYPE	SYNTAX	CATEGORY
Interpolation Property Attribute Class Style	<pre>{{expression}} [target]="expression"</pre> 	One-way from data source to view target
Event	<pre>(target)="statement"</pre> 	One-way from view target to data source
Two-way	<pre>[(target)]="expression"</pre> 	Two-way

@Input

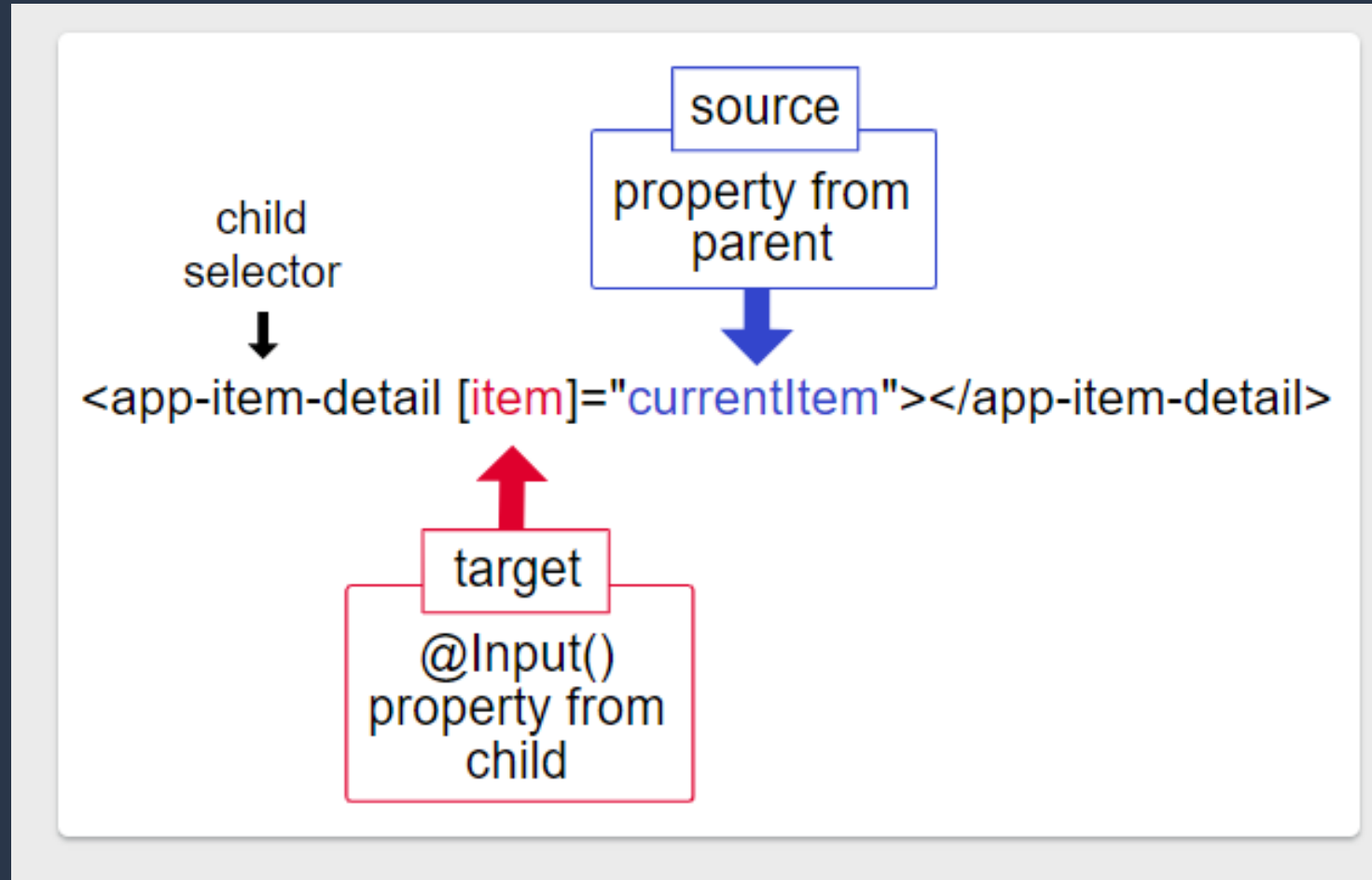
Sending data to a child component ↔

The `@Input()` decorator in a child component or directive signifies that the property can receive its value from its parent component.

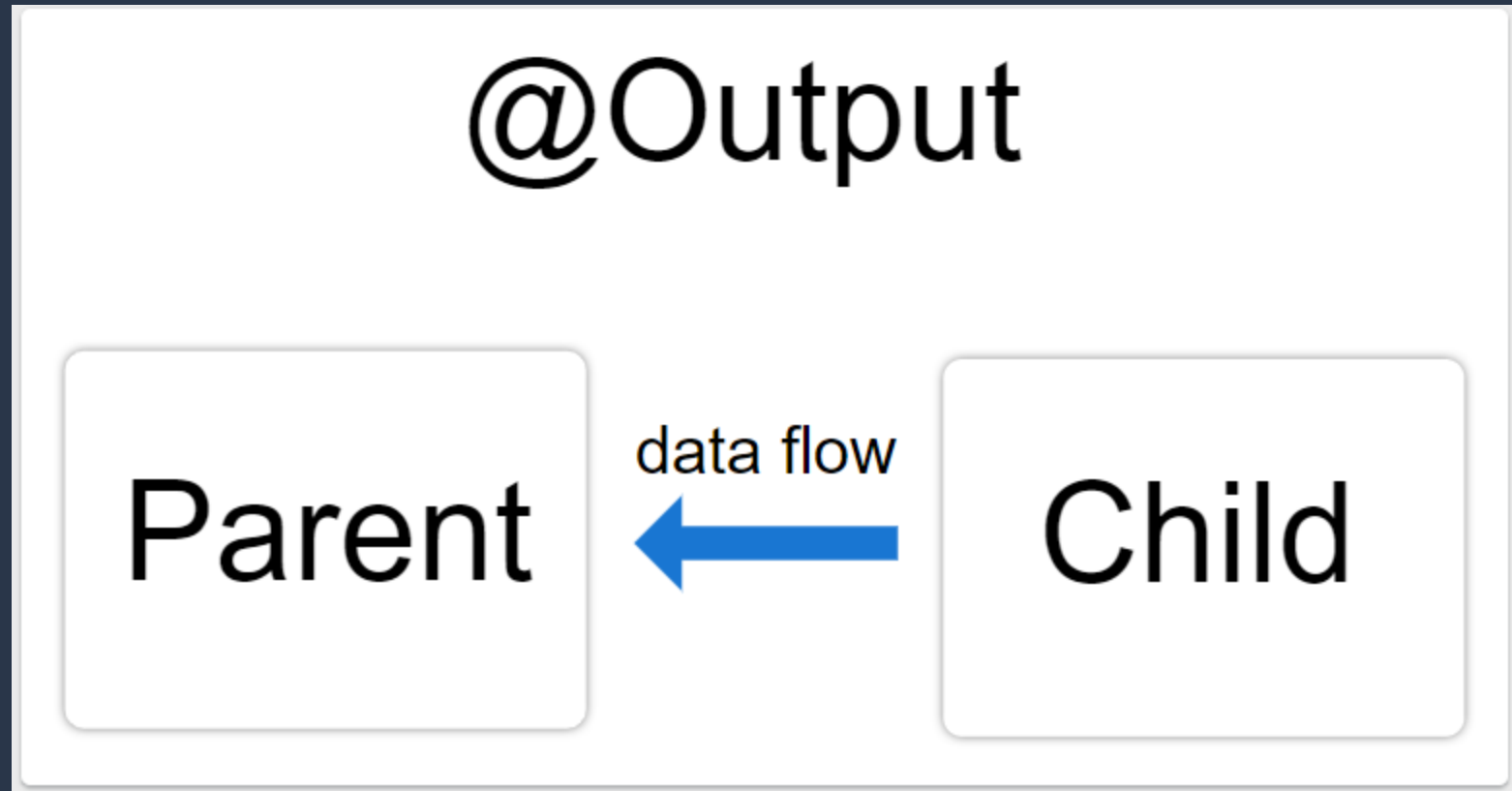


To use `@Input()`, you must configure the parent and child.

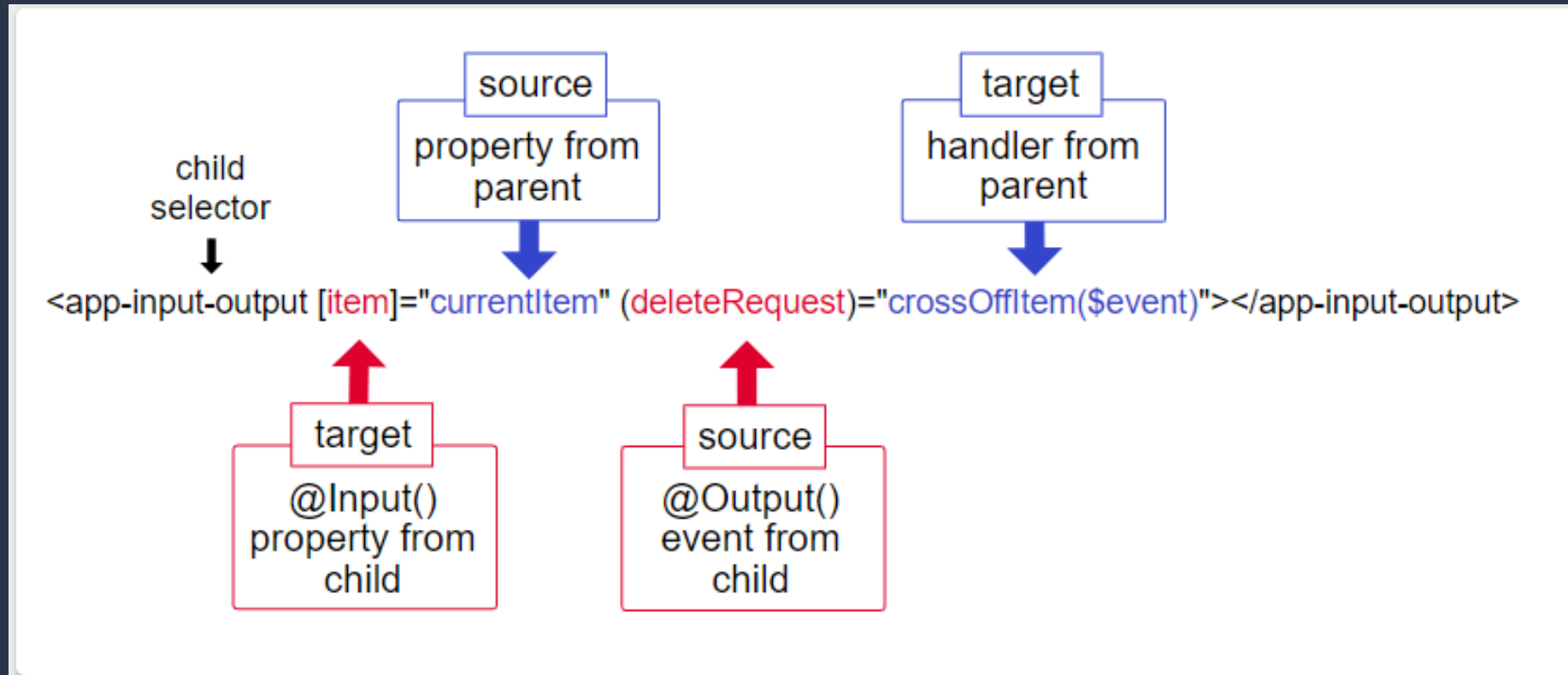
@Input syntax



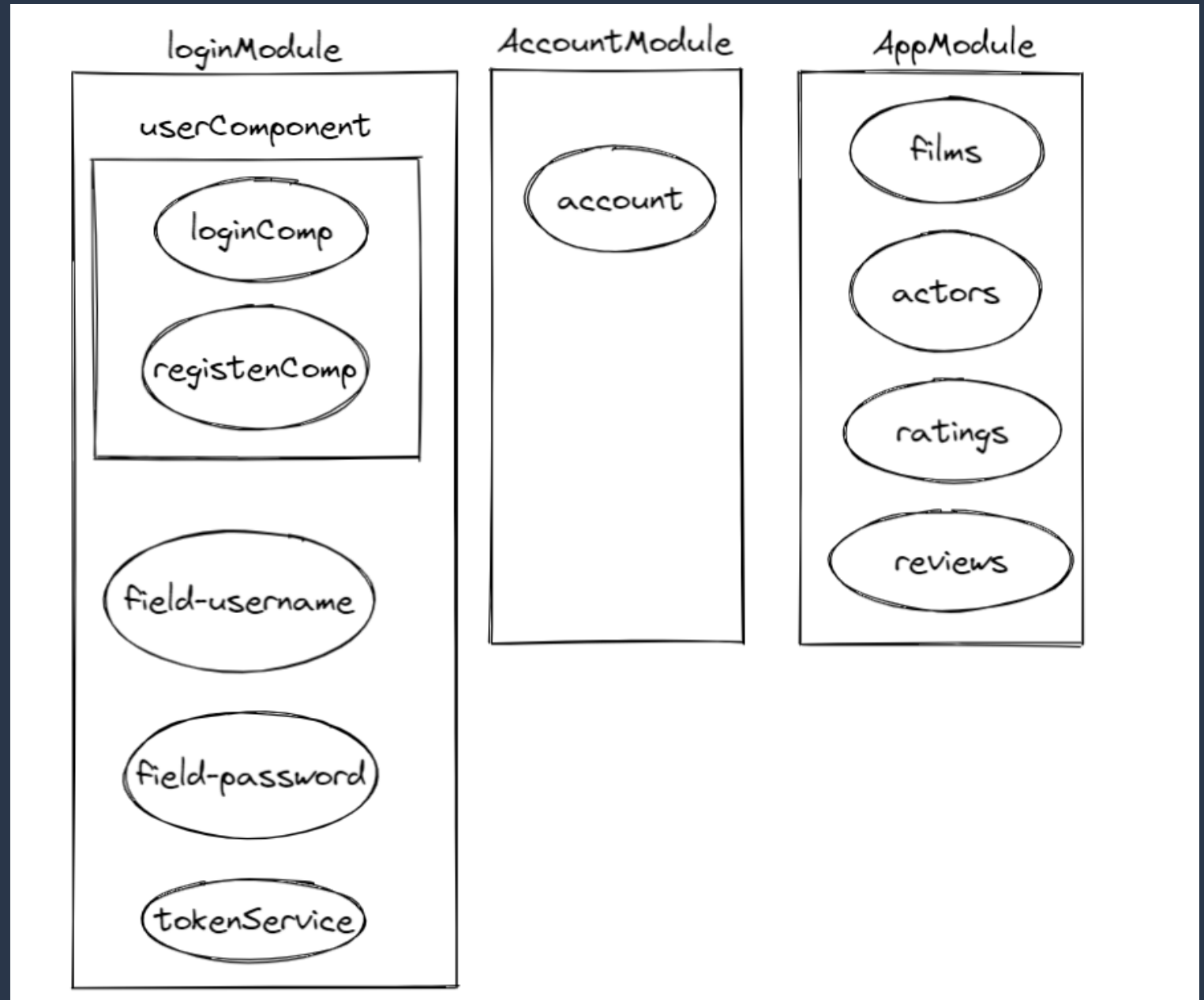
@Output syntax



Putting it all together



Films App



Task time

- Write an new field component : field-username
- Encapsulate property : maxLength
- Open properties to parent with @input : Lable and placeholder
- @Output event onBlur from child component, write to console the value of the username from the parent component (use emit in the child component)

Angular directives

- **Structural Built-in directives**

- *ngFor
- *ngIf
- ngSwitch

```
<div
  *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById"
  [class.odd]="odd">
  ({{i}}) {{hero.name}}
</div>

<ng-template ngFor let-hero [ngForOf]="heroes"
  let-i="index" let-odd="odd" [ngForTrackBy]="trackById">
  <div [class.odd]="odd">
    ({{i}}) {{hero.name}}
  </div>
</ng-template>
```

When structural directives are applied they generally are prefixed by an asterisk, *, such as *[ngIf](#). This convention is shorthand that Angular interprets and converts into a longer form. Angular transforms the asterisk in front of a structural directive into an [<ng-template>](#) that surrounds the host element and its descendants.

- **Attribute Built-in directives**

- ngStyle
- ngClass

Angular attr custom directives

```
@Directive({
  selector: '[appMyColor]'
})
export class MyColorDirective implements OnInit {
  @Input() txtColor : string = 'red';
  //@Input('appMyColor') txtColor : string = 'red';
  @HostBinding('style.backgroundColor') backgroundColor : string = 'gold';
  @HostBinding('style.color') color : string;

  constructor(private element: ElementRef) {}

  ngOnInit(): void {
    this.color = this.txtColor;
  }

  @HostListener('mouseenter') mo(){
    //this.element.nativeElement.style.backgroundColor = 'green';
    this.backgroundColor = 'green';
    //this.color = 'gold';
  }
}
```

```
<span appMyColor [txtColor]='''purple'''>Welcome</span>
```

```
<span appMyColor txtColor="purple">Welcome</span>
```

```
<span [appMyColor]='''red''' >Welcome</span>
```

```
import { Directive, ElementRef, OnInit, HostListener, HostBinding, Input, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appMyColorWithRenderer]'
})
export class MyColorWithRedererDirective implements OnInit {
  @Input() txtColor : string = 'red';
  constructor(private element: ElementRef, private renderer: Renderer2) {
  }
  ngOnInit(): void {
    this.renderer.setStyle(this.element.nativeElement, 'background-color', this.txtColor);
    this.renderer.setStyle(this.element.nativeElement, 'color', 'white');
  }
}
```


Angular structural custom directives

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appUnless]'
})
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set appUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

```
<p *appUnless="condition">
  paragraph content
</p>
```

Pipes : Build in pipes

- CurrencyPipe
- DatePipe
- DecimalPipe
- JsonPipe
- LowerCasePipe
- UpperCasePipe
- PercentPipe
- SlicePipe
- AsyncPipe

****Chain pipes left to right**

{{value | uppercase | date: 'fullDate'}} // not valid

{{value | date: 'fullDate' | uppercase }} // valid

```
persons.push(new Person("Dor", 34, 250.2, new Date("1988-06-22"), "123456789"));
```

```
<div
[ngStyle]="{'background-color': person.age === 34 ? 'green' : 'yellow' }"
[ngClass]="{'text-white': person.age === 34}">
  <ul>Id : {{person.id | getLast: 4}}</ul>      dor.zilka, 4 days ago • pi
  <ul>Name : {{person.name | uppercase}}</ul>
  <ul>Age : {{person.age }}</ul>
  <ul>Balance : {{person.balance | currency: 'USD'}}</ul>
  <ul>Birthday : {{person.balance | date: 'd/M/y'}}</ul>
</div>
```

Id : 6789

Name : DOR

Age : 34

Balance : \$250.20

Birthday : 1/1/1970

Pipes : Costume pipes

```
<ul>Id : {{person.id | getLast: 4}}</ul>
```

```
import {Pipe, PipeTransform} from "@angular/core";

@Pipe({
  name: 'getLast'
})

export class GetLast implements PipeTransform {
  transform(valueStr: string, last: number) {
    if(last > valueStr.length){
      return valueStr;
    }else{
      return valueStr.substring(valueStr.length - last , valueStr.length)
    }
  }
}
```

```
import { GetLast } from "../pipes/getLast";

@NgModule({
  declarations: [
    AppComponent,
    FirstComponent,
    UnlessDirective,
    MyColorDirective,
    GetLast
  ],
```

Pipes : Costume pipes

```
<li *ngFor="let person of persons | filterByAge: filterByAge">
```

dor.zilka, 4 days ago | 1 author (dor.zilka)

```
import {Pipe, PipeTransform} from '@angular/core'  
import { Person } from "../models/person";
```

dor.zilka, 4 days ago | 1 author (dor.zilka)

```
@Pipe({  
  name: 'filterByAge'  
})
```

```
export class FilterByAge implements PipeTransform{  
  transform(persons: Array<Person>, ageFiler: number) : Array<Person> {  
    var result : Array<Person> = persons.filter(p=> p.age === ageFiler)  
    return result.length == 0 ? persons : result;  
  }  
}
```

Pipes : async pipes

```
<h3>{{asyncVal}}</h3>  
<h3>{{asyncVal | async}}</h3>
```

[object Promise]

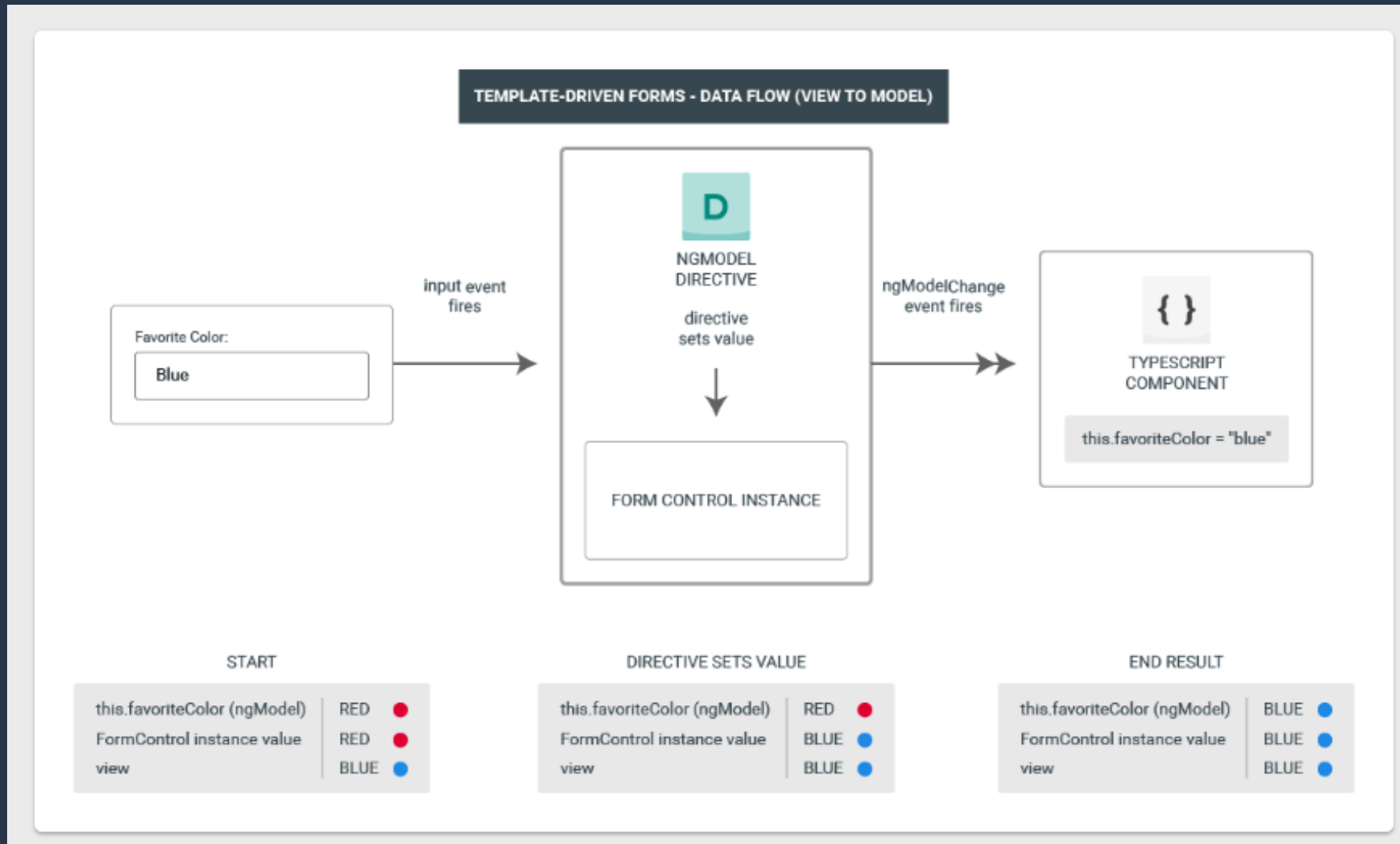
Hello im async pipe

```
asyncVal = new Promise((resolve,reject) =>{  
  |   setTimeout(() => {  
  |     resolve('Hello im async pipe');  
  |   }, 3000);  
  | })
```

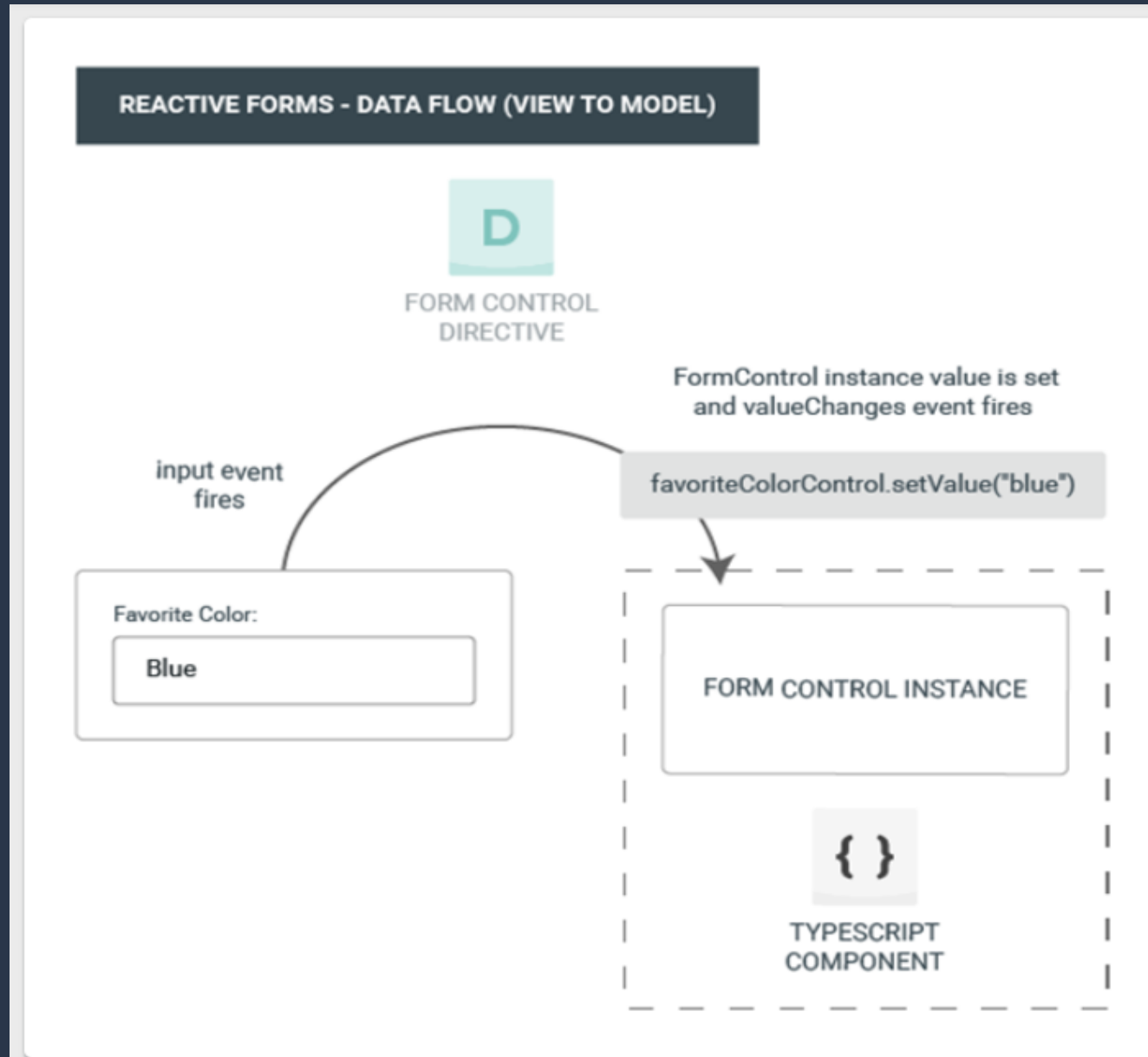
Angular Forms : Reactive VS Template

	REACTIVE	TEMPLATE-DRIVEN
Setup of form model	Explicit, created in component class	Implicit, created by directives
Data model	Structured and immutable	Unstructured and mutable
Data flow	Synchronous	Asynchronous
Form validation	Functions	Directives

Template driven : Data flow (async)



Reactive form : Data flow (sync)



Forms : Template driven

- Review login form in films
 - Form & ngForm
 - ngSubmit
 - [disabled]
 - ngModel
 - ng-dirty ng-valid ng-touched ng-pristine
 - Use refElements validation
 - ngModelGroup
 - Get controls values (key:value)

```
<h1>Login Form</h1>
<form (ngSubmit)="onSubmit(loginForm)" #loginForm="ngForm">
  <div ngModelGroup="userData">
    <label for="name">Name</label>
    <input required #name="ngModel"
      id="name"
      ngModel
      name="name"
      class="form-control"
    >
    <div [hidden]="name.valid || name.pristine" >
      Name is required
    </div>
  </div>
  <br/>
  <label for="name">Last name</label>
  <input required #lName="ngModel"
    id="lName"
    ngModel
    name="lName"
    class="form-control"
  >
  <div [hidden]="lName.valid || lName.pristine" >
    Last Name is required
  </div>
</div>
<br/>
<label for="platform">platform</label>
<select #platform="ngModel" required
  ngModel
  name="platform"
  class="form-control">
  <option *ngFor="let x of ['facebook','gmail','github']" [value]="x"> {{x}}</option>
</select>
<div [hidden]="platform.valid || platform.pristine">
  Platform is required
</div>
<br/><br/>
<button type="submit" [disabled]="!loginForm.valid">Submit</button>
</form>
<div [hidden]="!loginForm.submitted">
  <h2>You Logged as : {{ name.value }}</h2>
</div>
```

Task

- Create a lead form
 - UserDetails
 - Name
 - Last name
 - Id
 - Phone number
 - Phone type
 - Area code (select)
 - Phone number
 - Address
 - City
 - Street
 - Submit
 - Disabled until valid
 - Console log the form