Angular

Első előadás

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Probléma (1)

- 2009-et írunk (2 évvel az iPhone után)
- Elkezdődik az átállás az okos kliens alkalmazásokról (*Silverlight*, *Flash*) vékonykliensekre
- A legelterjedtebb kliensoldali keretrendszer a **JQuery**, ami DOM-manipulációra lett kifejlesztve
  - > A DOM-manipuláció lassú
  - > Az adatot nem illik a DOM-ban tárolni

MV* (1)

- Model
- View
- *
  - > Controller
  - > ViewModel
  - > Presenter
  - > ...

Komponensek

- A felhasználói felület egy részletéért felelős
  - > Újrahasznosítható
- A felület leírását **deklaratívan**, HTML-szerű **sablonokkal** adjuk meg
  - > A **sablon** egy nézetet és annak állapotát, UI kezelő logikáját tartalmazza
  - > A sablon lehet HTML vagy SVG is!
- Bemeneti **adatokat** vár, kimenő **eseményeket** publikál (`@Input`, `@Output`)
  - > Az **adatkötéseket** a sablonban vannak elhelyezve
- Hivatkozhat más komponensekre saját sablonjában
  - > Ezáltal egy **komponens-hierarchia** épül fel

```
export class ProductAlertsComponent {
  @Input() product: Product | undefined;
  @Output() notify = new EventEmitter();
}
```


Direktívák

- Direktívák típusai:
  - > **Attribútumdirektíva**: adott DOM elemhez kiegészítő működést vagy megjelenést rendel
  - > **Strukturális direktíva**: Megváltoztathatja a DOM struktúráját
- **Eseménykezelőket** regisztrálhatunk a DOM elemre
- **Adatkötéshez** használhat `@Input` paramétereket és publikálhat `@Output` eseményeket
- A direktíva csak egy osztály
  - > **Nem komponens, mert nincs felülete**

Alkalmazás-életciklus

- Az angular alkalmazás = komponensek, direktívák életciklusát a keretrendszer vezérli
- A DI segítségével a sablonok feldolgozása után példányosítja a komponenseket, azok függőségeit
  - > Később aszinkron események hatására új komponenseket, függőségeket gyárt

Angular Module / NgModule

- Egy alkalmazás önmagában összefüggő része
  - > Komponensek, direktívák, ...
  - > Újrafelhasználás és/vagy modularizálás
- A keretrendszer maga is moduláris, mindig csak azokat a modulokat használjuk, amire szükségünk is van:
  - > Core – mindig kell
  - > Common – nagyon gyakori
  - > Forms – űrlapok készítése
  - > Http – http kommunikáció
  - > Router - útvonalválasztás
  - > ...

Csővezetékek

- A pipe (csővezeték) a felületen megjelenő adat formázására szolgál
- Néhány beépített pipe lehet segítségünkre:
  - > DatePipe
  - > UpperCasePipe/LowerCasePipe

Probléma (1)

- 2009-et írunk (2 évvel az iPhone után)
- Elkezdődik az átállás az okos kliens alkalmazásokról (*Silverlight*, *Flash*) vékonykliensekre
- A legelterjedtebb kliensoldali keretrendszer a **JQuery**, ami DOM-manipulációra lett kifejlesztve
  - A DOM-manipuláció lassú
  - Az adatot nem illik a DOM-ban tárolni

```
jQuery(document).ready(function(jQuery) {

    // Prevent the enter key from submitting the form.
    jQuery(".ninja-forms-form input").bind("keypress", function(e) {
        if (e.keyCode == 13) {
            var type = jQuery(this).attr("type");
            if( type != "textarea" ){
                // return false;
            }
        }
    });

    /* * * Begin Mask JS * * */

    jQuery("div.label-inside input, div.label-inside textarea").focus(function(){
        var label = jQuery("#" + this.id + "_label_hidden").val();
        if( this.value == label ){
            this.value = '';
        }
    });

    jQuery("div.label-inside input, div.label-inside textarea").blur(function(){
        var label = jQuery("#" + this.id + "_label_hidden").val();
        if( this.value == '' ){
            this.value = label;
        }
    });

    if( jQuery.fn.mask ){
        jQuery(".ninja-forms-mask").each(function(){
            var mask = jQuery(this).data('mask');
            mask = mask.toString();
            jQuery(this).mask(mask);
        });

        var date_format_mask = ninja_forms_settings.date_format;
        date_format_mask = date_format_mask.replace( /m/g, 9 );
        date_format_mask = date_format_mask.replace( /d/g, 9 );
        date_format_mask = date_format_mask.replace( /y/g, 99 );
```

Probléma (2)

- A **JavaScriptben** írt alkalmazások **kódbázisa** jelentősen **nőni** kezdett
- **Nem voltak elterjedt, egységes konvenciók** az alkalmazásfejlesztés területén
- A vastagkliens- és szerveralkalmazásokban viszont kellően komplex architekturális mintákat alkalmaztak
  - MVVM, MVP, MVC

Probléma (3)

- Az alkalmazás növekedésével lehetetlen volt a JavaScript dinamikussága és a DOM-manipuláció miatt látni egy apró módosítás hatását az alkalmazásra

Az így készített alkalmazások hosszú távon nem fenntarthatóak

Megoldás

- 2010, a Google megalkotja az AngularJS alkalmazásfejlesztő keretrendszert
  - Elsősorban (de nem kizárólag) SPA alkalmazások fejlesztésére
  - Voltak korábban is próbálkozások, de a kisebb keretrendszerek nem terjedtek el

Alapelvek

- Teljes értékű keretrendszer
- Open-source
- Deklaratív UI leírás
- Imperatív üzleti logikai leírás
- DOM-manipulációs logika leválasztása az alkalmazáslogikától
- Kliens- és szerveroldali fejlesztések párhuzamosítása
- Separation of concerns
- Dependency Injection

Újabb probléma

- 2014, a legelterjedtebb vékony kliens keretrendszer az AngularJS
- Az idő közben bekövetkezett paradigmaváltások és az AngularJS néhány alapkoncepciója kérdésessé teszi a keretrendszer jövőjét

Újabb megoldás

- 2016, megjelenik az Angular 2.0
  - Teljes újraírás, nem visszafelé kompatibilis
  - Paradigmaváltások
    - **TypeScript** alapon
    - Kizárólag **modern böngészőket** támogat (IE9+)
    - Reaktív felületek készítésének lehetősége RxJS-sel
    - **Egyszerűbb templating**
    - **Egyértelműbb architektúra**
    - Nincs többé $scope

Elnevezések

- 1.0: AngularJS
- 2.0+: Angular
  - A 3.0-as verziót a Router komponens verzióugrása miatt kihagyták
  - Ütemezett kiadási ciklus:
    - Kb. 6 havonta új főverzió
    - 1-3 alverzió főverzióként
    - Akár heti sűrűségű bugfix kiadás

Fejlesztőeszközök



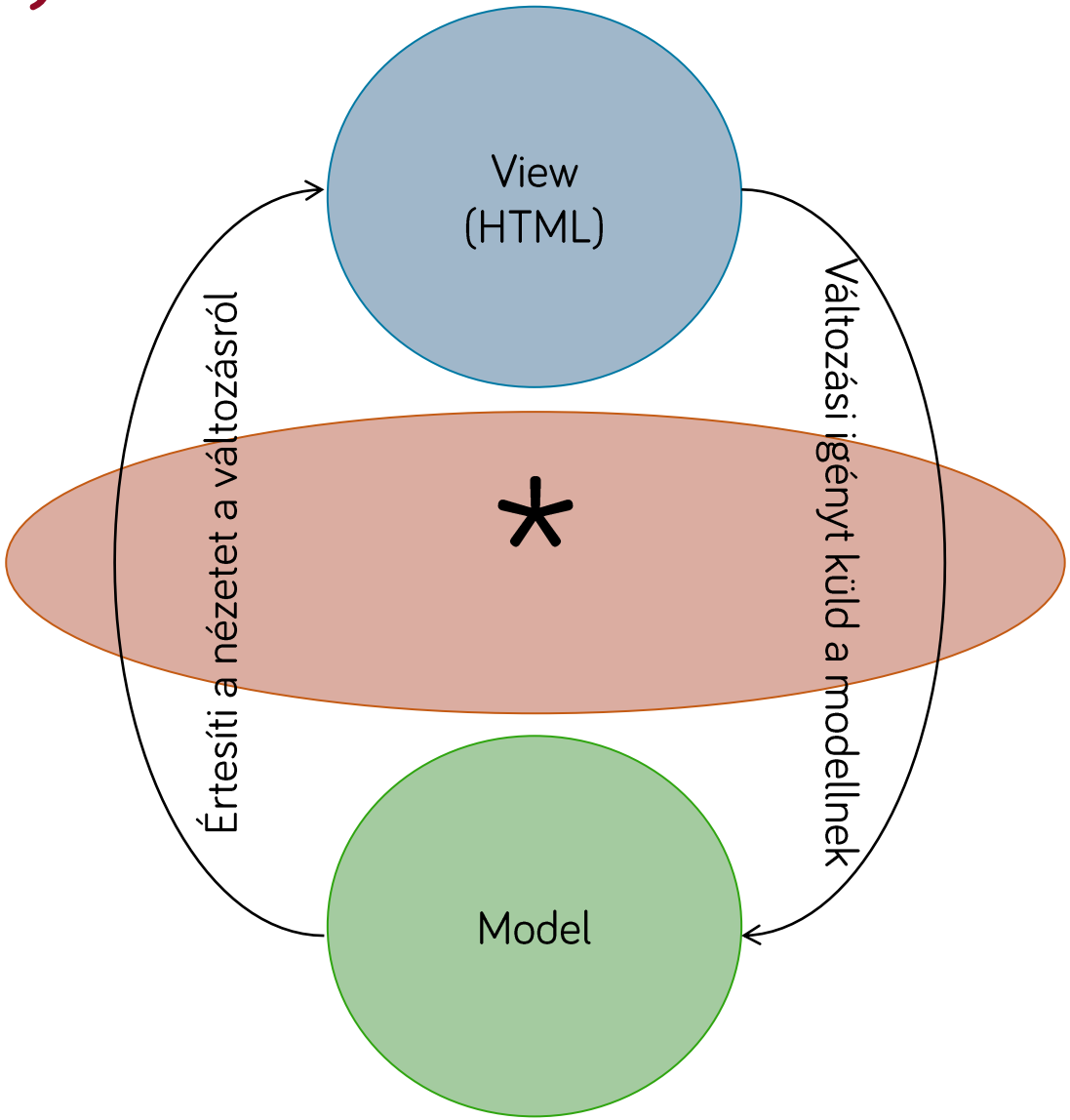
. . .

- **npm**: csomagkezelés
- **@angular/cli**: **ng** parancssori eszköz
  - Kódgenerálás
  - Kódoptimalizálás (bundling, minification)
  - Hibakeresés lokális fejlesztői szerveren

MV* (1)

- Model
- View
- *
  - Controller
  - ViewModel
  - Presenter
  - ...

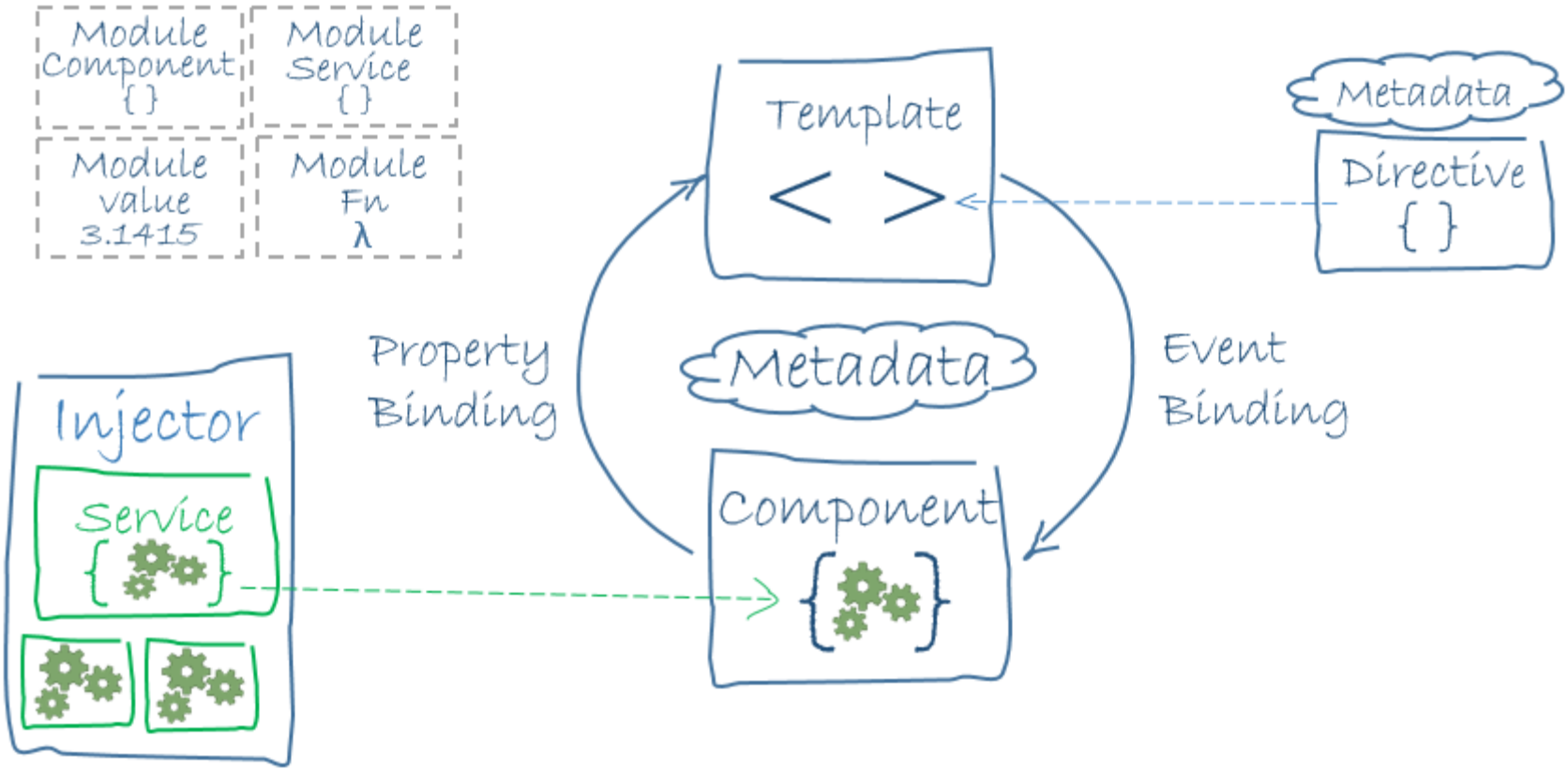
MV* (2)


View
(HTML)
*
Model
Értesíti a nézetet a változásról
Változási igényt küld a modellnek

MV* (3)

- Lehetővé teszi az alkalmazáslogika, a nézet összeállításának és a kettő közötti kommunikáció szétválasztását
- Az egyes (részben automatizált) kommunikációs lépéseket a Model és a View között mindkét irányban adatkötésnek nevezzük

Angular architektúra



https://angular.io/guide/architecture

Metaadatok

- Az alkalmazás egyes részeinek **plusz információval** kell szolgálniuk az Angular felé, ami egyszerű típusinformációból nem következik
  - Ezeket **metaadat**ként adjuk meg
- A **TypeScript dekorátorokkal** plusz információt rendelhetünk:
  - Egy osztályhoz
  - Egy függvényhez vagy konstruktorhoz
  - Egy függvényparaméterhez
  - Egy tulajdonsághoz vagy accessorhoz

Angular dekorátorok

- Az Angular alkalmazásban gyakran használunk dekorátorokat, amik így plusz információt (AOP) hordoznak:
  - @NgModule
  - @Component, @Directive
  - @Input, @Output
  - @Inject, @Injectable

Komponensek

- A felhasználói felület egy részletéért felelős
  - Újrahasznosítható
- A felület leírását **deklaratívan**, HTML-szerű **sablon**okkal adjuk meg
  - A **sablon** egy nézetet és annak állapotát, UI kezelő logikáját tartalmazza
  - **A sablon lehet HTML vagy SVG is!**
- Bemeneti **adatok**at vár, kimenő **esemény**eket publikál (`@Input`, `@Output`)
  - Az **adatkötéseket** a sablonban vannak elhelyezve
- Hivatkozhat más komponensekre saját sablonjában
  - Ezáltal egy **komponens-hierarchia** épül fel

```
export class ProductAlertsComponent {
  @Input() product: Product | undefined;
  @Output() notify = new EventEmitter();
}
```

Komponensek és szolgáltatások

- A komponensek jól körülhatárolt funkciót valósítanak meg az alkalmazásban
- Minden komponenshez tartozik egy nézet
  - > = sablon = template
- A komponens **felhasználja** az alkalmazás modelljét (üzleti logika) **szolgáltatások** formájában, az **nem** a komponensben van megírva

```
addToCart(product: Product) {
  this.cartService.addToCart(product);
  window.alert('Your product has been added to the cart!');
}
```

Hello Angular alkalmazás - Component

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


@Component({
  selector: 'hello-component',


  template: `<h1>Hello {{title}}!</h1>`
})
export class MyComponent {
  title = 'Angular';
}
```

A Component dekorátor, amit az Angular definiál

A komponens dekorátorral metaadat kötése a komponensünkhöz

A CSS selector, amire a komponensünk példányosodik

A saját komponens osztályunk állapottárolásra (és eseménykezelésre)

A komponensünk HTML template-je **adatkötéssel**

Komponens felhasználása

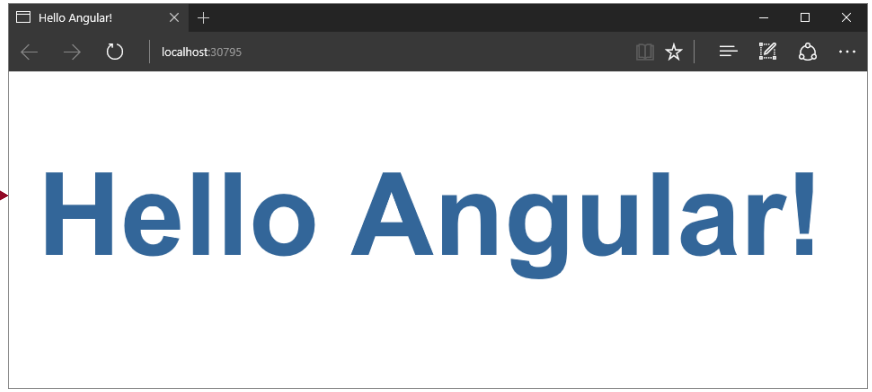
```
<body>
  <hello-component>Loading...</hello-component>
</body>
```

1. Az alkalmazás indulása után a komponensek illesztése

```
@Component({
  selector: 'hello-component',
  template: `<h1>Hello {{title}}!</h1>`
})
export class HelloComponent {
  title = 'Angular';
}
```

2. A kezdeti adatkötések végrehajtása

3. Renderelés a DOM-ba



Szöveg kiírás

- Kifejezés értékének kiírása a HTML-be
- `{{ … }}`
- A zárójeles kifejezés kiértékelődik és szövegesen megjelenik
  - Lehet property neve vagy bármilyen kifejezés
  - A kifejezés kontextusa a komponens: a „title” a komponens propertyje -> a nézetet kötjük a modellhez

```
<h1>Hello {{title}}!</h1>

<p>The sum of 1 + 1 is {{1 + 1}}.</p>
```

Adatkötés

- HTML elemek, komponensek tulajdonságainak beállítása
- `[…] = "…"`
  - Az **idézőjelben** lévő **kifejezés kiértékelődik**
  - A kontextus a komponens -> a nézetet kötjük a modellhez

`<img [src]="itemImageUrl">`: *HTML DOM property*

`<component [prop]="data">`: *komponens, direktíva @Input*

- Ekvivalens az

  `<elem directive="{{data}}">`

  szintaxissal!

Adatkötés `@Input` dekorátorral

- Komponenseken (és direktívákon) az osztály változóit kívülről állíthatjuk
- **@Input** dekorátorral kell megjelölni a mezőt vagy tulajdonságot

```
@Input() title: string = 'Angular';
```

  - Ahogy egy HTML objektumnak állítjuk a tulajdonságai, úgy állítjuk egy komponens inputjait

```
<app-hello [title]="name"></app-hello>
```

- A komponens szülője így dinamikusan paramétert tud átadni a sablonban a gyereknek

Feladat: szín választó

Adatkötés @Input dekorátorral (2)

```
@Component({
  selector: 'app-color-picker',
  template: `<div>
              <input type="radio" name="colors"
                     (click)="color='lightgreen'">Green
              <input type="radio" name="colors"
                     (click)="color='yellow'">Yellow
             </div>
             <show-value [value]="color"></show-value>`
})
export class ColorPickerComponent {
  color: string;
}
```

Inline utasítás


```
@Component({
  selector: 'show-value',
  template: `<h1>{{value}}</h1>`
})
export class ShowValueComponent {
  @Input()
  value: string;
}
```

Egyirányú adatkötés

CSS class kötése

- Egyetlen osztály

```
<td [class.centered]="isCentered">közép</td>
```

  > Kiértékeli az isCentered kifejezést -> boolean

- Több osztály

```
<td [class]="expr">hosszú szöveg</td>
```

  > Szöveg, **felsorolt** osztályok: "activeRow centered"

  > Szöveg **tömb**: ["activeRow", "centered"]

  > **Kulcs-érték** párok: {activeRow: true, centered: false}

CSS stílusok kötése

- Egyetlen stílus

```
<td [style.text-align]="expr">közép</td>
```

```
<td [style.textAlign]="expr">közép</td>
```

  - expr = "center"

- Több stílus

```
<td [style]="expr">közép</td>
```

  - Szöveg:
    expr = "text-decoration: underline; text-align: center;"
  - Kulcs-érték párok:
    expr = {textDecoration: "underline", textAlign: "center"}

Események kötése

- Esemény kezelő kötése
- (…) = "…"
- A **zárójeles** eseményre **végrehajtódik** az idézőjeles **utasítás** (statement)
  - Tipikusan a komponens egy metódusának **meghívása**
  - A kontextus a sablon és a komponens példány, globális névtér nem érhetőek (például console.log stb.)
  - Csak minimális műveleteket végezzünk!

```
<button type="button" (click)="onSave()">Save</button>
```

- Billentyűzet események támogatása

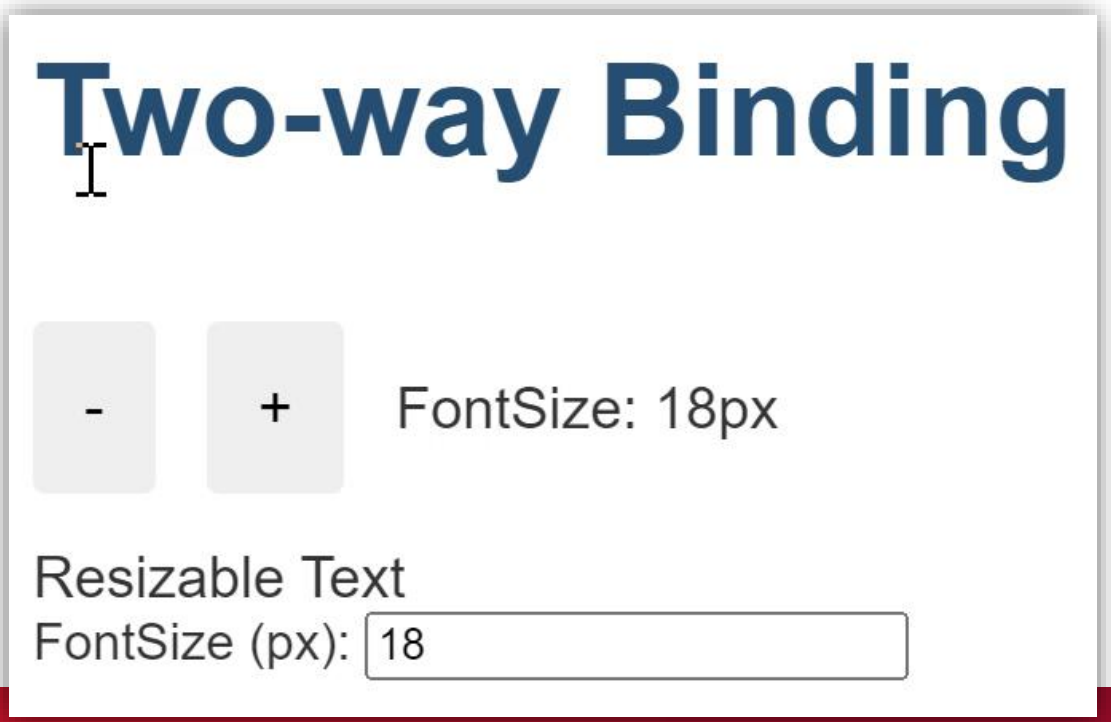
```
<input (keydown.shift.t)="onTravel()"/>
```

Kétirányú adatkötés

- Egyszerre állítunk be értéket és figyeljük a változását
  - A gyerek komponens **olvassa és írja** is az értéket
  - Property adatkötés `[ ]` + eseménykötés `( )`
- Szintaktika: `[( var )]`
- Gyerek komponens
  - `@Input` a *var* mezőn vagy propertyn
  - `@Output` a *varChange* eseményen
- Szülő komponens a saját változóját köti a *var*-hoz

Kétirányú adatkötés példa

- Gyerek komponens:
  - Nyomógombok + / - és mellette a FontSize label
- Szülő
  - Saját fontSizePx változó
  - Input mező



Kétirányú adatkötés „gyerek” kód

```
<div>
  <button type="button" (click)="dec()" title="smaller">-</button>
  <button type="button" (click)="inc()" title="bigger">+</button>
  <span [style.font-size.px]="size">FontSize: {{size}}px</span>
</div>
```

```
export class SizerComponent {

  @Input()  size!: number | string;
  @Output() sizeChange = new EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

Kétirányú adatkötés „szülő” kód

```
<div id="two-way-1">
  <app-sizer [(size)]="fontSizePx"></app-sizer>
  <div [style.font-size.px]="fontSizePx">Resizable Text</div>
  <label>FontSize (px): <input [(ngModel)]="fontSizePx"></label>
</div>
```

```
export class AppComponent {
  fontSizePx = 16;
}
```

Adatkötés összefoglalás

- Forrás
  - **Sablon**, például nevesített tagek stb.
  - **Komponens** példány
- Cél
  - **HTML** property vagy attribútum
  - CSS stílusok
  - Komponens, direktíva **@Input**tal megjelölt változója
- **Kifejezés**, amit kiértékel a rendszer
  - {{*kif*}} -> kifejezés kiértékelése
  - [param] = "*kif*" -> kifejezés kiértékelése
  - (e) = "*utasítás*" -> utasítás végrehajtása

Attribútum vs property

- HTML **tegnek** van **attribútuma**, ami a DOM létrehozáskor inicializálja a HTML **objektum propertyjét**
  - Majdnem mindegyik attribútumhoz tartozik property, azonos vagy hasonló névvel
    - td . colspan: attribútum
    - td . colSpan: property
- A [*tag*] szintaktika propertyt állít
- Attribútum állítása: [attr . *tag*]

```
<td [attr.colspan]="1+1">hosszú szöveg</td>
```

Direktívák

- Direktívák típusai:
  - **Attribútumdirektíva**: adott DOM elemhez kiegészítő működést vagy megjelenést rendel
  - **Strukturális direktíva**: Megváltoztathatja a DOM struktúráját
- **Eseménykezelők**et regisztrálhatunk a DOM elemre
- **Adatkötés**hez használhat `@Input` paramétereket és publikálhat `@Output` eseményeket
- A direktíva csak egy osztály
  - **Nem komponens, mert nincs felülete**

Attribútumdirektívák

- Az attribútumdirektíva nem cseréli le az elemet a DOM-ban, hanem megváltoztatja, kiegészítő működést rendel hozzá
- Beépített attribútumdirektívák:
  - `[ngStyle]`: stílus, ugyanaz, mint az [style]
  - `[ngClass]`: osztály, ugyanaz, mint a [class]
  - `[(ngModel)]`: modell kétirányú kötése egy (input) elemhez

Attribútum direktíva - példa

- A HTML elem tulajdonságát közvetlenül állítja

```
import { Directive, ElementRef } from '@angular/core';

@Directive( { selector: '[appBold]' } )
export class BoldDirective
{
  constructor( el: ElementRef )
  {
    el.nativeElement.style.fontWeight = 'bold';
  }
}
```

```
<p appBold>Lorem ipsum...</p>
```

Attribútum direktíva – HostListener

- Eseményekre tudunk feliratkozni a **HostListenerrel**

```
@Directive({
  selector: '[appBold]',
})
export class BoldDirective {
  private el: ElementRef;

  constructor(el: ElementRef) {
    this.el = el;
  }

  @HostListener('mouseenter') onMouseEnter() {
    this.el.nativeElement.style.fontWeight = 'bold';
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.el.nativeElement.style.fontWeight = 'normal';
  }
}
```

Attribútum direktíva – adatkötés

- @Input(...) lehetővé teszi az adatkötést
- Meg kell adni az input paraméter nevét!
  - Ha a direktíva selectorának a nevét adjuk meg, akkor az az alapértelmezett

```
@Input('appBold') bold = false;
```

```
<p [appBold]='true'>Lorem ipsum...</p>
```

  - Ha több paraméter van, akkor más neveket kell megadni, ügyelve a névütközésre!

Attribútum direktíva – több paraméter

```
@Directive({ selector: '[appBold]' })
export class BoldDirective {
  @Input('appBold') bold = false;
  @Input('BoldWeight') weight = 300;
  private el: ElementRef;

  constructor(el: ElementRef) {
    this.el = el;
  }
  @HostListener('mouseenter') onMouseEnter() {
    this.el.nativeElement.style.fontWeight =
                  this.bold ? this.weight : 'normal';
  }
  @HostListener('mouseleave') onMouseLeave() {
    this.el.nativeElement.style.fontWeight = 'normal';
  }
}
```

```
<p [appBold]='true' [BoldWeight]=600>Lorem ipsum...</p>
```

Lorem ipsum...

FYI: hiba

- Az alábbi kód **fordítási** hibát dob

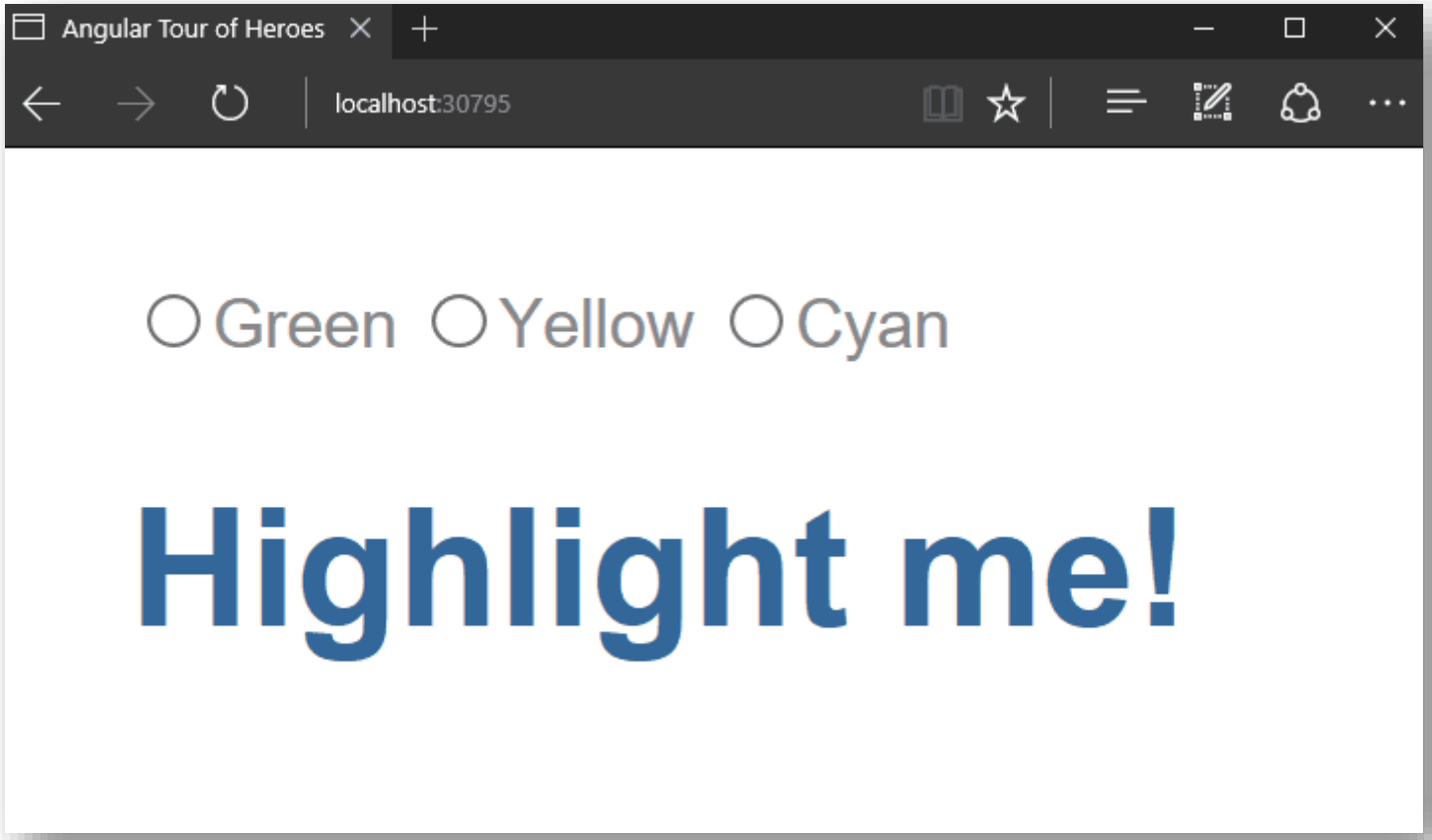
```
<p [appBold]>Lorem ipsum...</p>
```

```
error TS2322: Type 'undefined' is not assignable to type 'boolean'.
```

- Megoldás

```
@Directive({ selector: '[appBold]' })
export class BoldDirective {
  @Input('appBold') bold: boolean | undefined = true;
```

Feladat: kijelölés állítható színnel



Kijelölés, kívülről állítható színnel

```
@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
  constructor(public el: ElementRef) { }

  @Input('myHighlight') highlightColor: string;

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

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

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

@Input dekorátor opcionális megnevezéssel

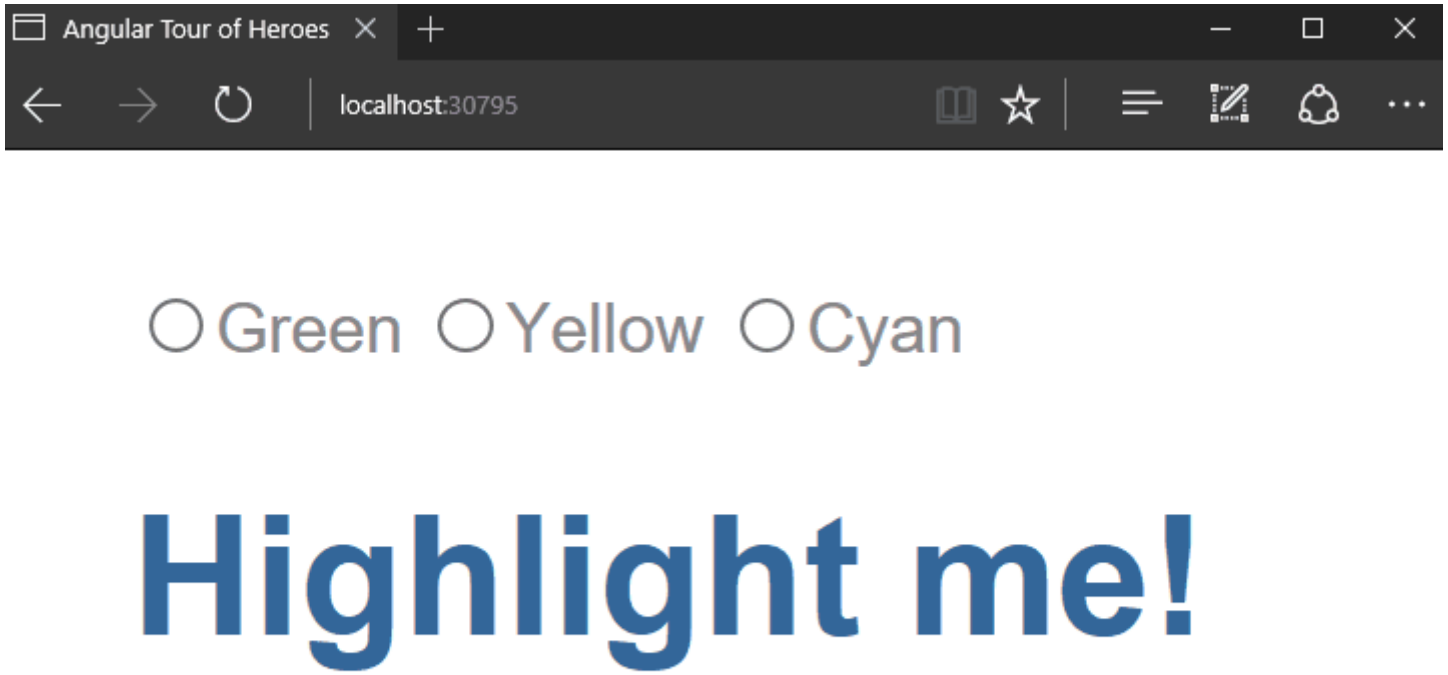
Szín választó

```
@Component({
  selector: 'app-color-picker',
  template: `<div>
              <input type="radio" name="colors"
                     (click)="color='lightgreen'">Green
              <input type="radio" name="colors"
                     (click)="color='yellow'">Yellow
              <input type="radio" name="colors"
                     (click)="setColor('cyan')">Cyan
             </div>
             <h1 [myHighlight]="color">Highlight me!</h1>`
})
export class ColorPickerComponent {
  color: string;
  setColor(color: string) {
    this.color = color;
  }
}
```

Egyirányú adatkötés (V→M)

Egyirányú adatkötés (M→V)

Adatkötés `@Input` dekorátorral



Esemény publikálása @Outputtal

```
@Directive({ selector: '[myClick]' })
export class MyClickDirective {
  @Output('myClick') clicks = new EventEmitter<void>();
  @HostListener('click') onClick() {
    this.clicks.emit();
  }
}
```

@Output dekorátor opcionális megnevezéssel, kívülről látható események publikálására

```
@Component({
  selector: 'hello-component',
  template: `<h1 (myClick)="clicks=clicks+1">
                Clicked {{clicks}} times!</h1>`
})
export class HelloComponent {
  clicks = 0;
}
```

Külső eseménykezelő feliratkoztatása

Clicked 0 times!

Adatkötés `@Output` dekorátorral

- A sablonból elérhető **$event** objektumot átadhatjuk paraméterül az eseménykezelőnek
  - Nem javasolt, túlságosan erős kötést jelent a nézet és a modell között

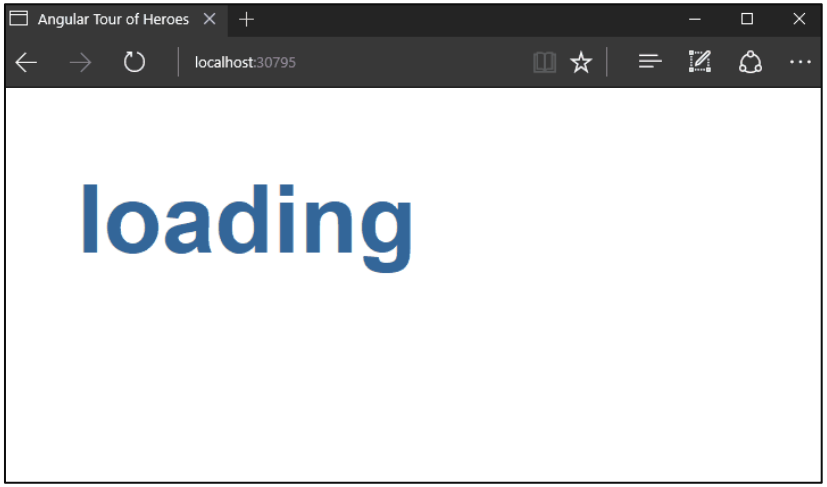
```
@Component({
  selector: 'hello-component',
  template: `<h1 (myClick)="myClickHandler($event)">
                  Clicked {{clicks}} times!</h1>`
})
export class HelloComponent {
  clicks = 0;
  myClickHandler($event: MouseEvent) { this.clicks++; }
}
```

Strukturális direktívák

- Olyan direktíva, ami a DOM-ot módosítja
- A * mikroszintaxis előzi meg az attribútumot
- Beépített strukturális direktívák:
  - ***ngIf**: megjeleníti az elemet a DOM-ban, ha a megadott feltétel teljesül
  - ***ngSwitchCase, *ngSwitchDefault** : adott kifejezés értékének vizsgálata, és a megfelelő feltétel esetén az adott template renderelése a DOM-ba
    - Az **[ngSwitch]** nem strukturális, mert csak egy logikai egységet alkot a case-ek és default körül
  - ***ngFor**: iterál egy adott tömbön, és a megadott DOM-elemet ismétli meg (adatkötésekkel)

Példa: loading felirat

- Feladat: a „loading” és „Hello Angular” feliratokat cserélgessük véletlenszerűen

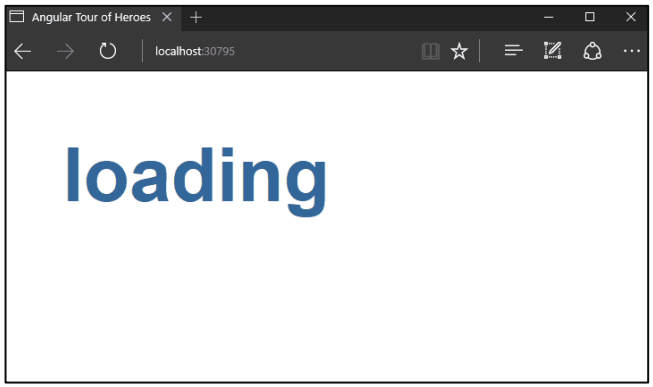


Strukturális direktíva - *ngIf

```
@Component({
  selector: 'hello-component',
  template: `
  <h1 *ngIf="title === '...'">loading</h1>
  <h1 *ngIf="title !== '...'">Hello {{title}}!</h1>
  `
})
export class HelloComponent {
  title: string = "...";
  constructor() {
    setInterval(() => this.title = Math.random() < 0.5
                                   ? "..." : "Angular", 1000);
  }
}
```

Az *ngIf teljesül, ha a title értéke "..."

Az *ngIf teljesül, ha a title értéke **nem** "..."



Másodpercenként véletlenszerűen beállítjuk az adatkötött title értéket

*ngIf: elem be- és kikapcsolása

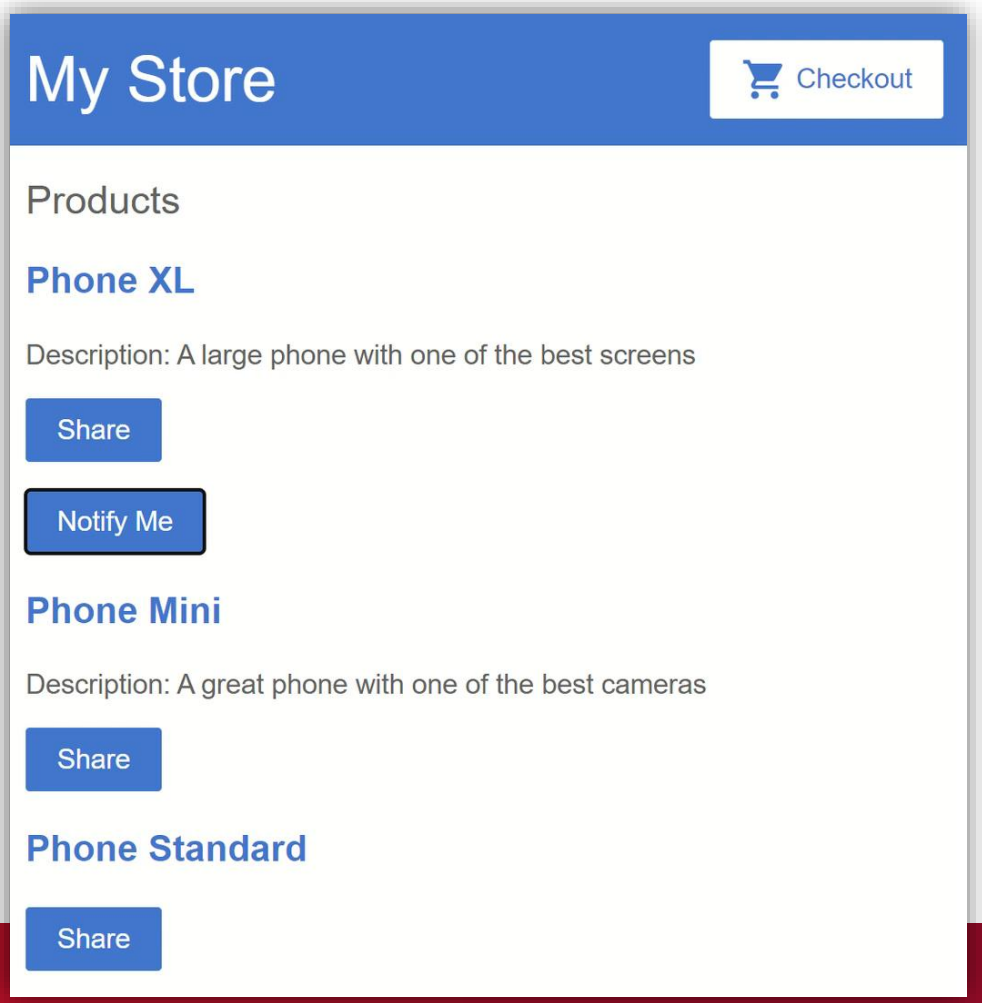
- Kiértékeli a kifejezést

```
<p *ngIf="numbers.length>3">Sok szám</p>
```

- Teljesen kiveszi az elemet
  - > Az elem tartalmát nem értékeli ki
  - > A teljes részfát törli, a komponensekkel együtt
- Ha csak rejteni akarjuk
  - > display-t kell none-ra állítani adatkötéssel
  - > Nincs külön megoldás, mint pl. Vue-ban
- Van ngSwitch, kényelmesebb, mint a sok ngIf

Példa: termék lista

- Listázzuk a termékeket
  - Névvel
  - Leírással
  - megosztás gombbal
  - ha drága, akkor értesítés kéréssel

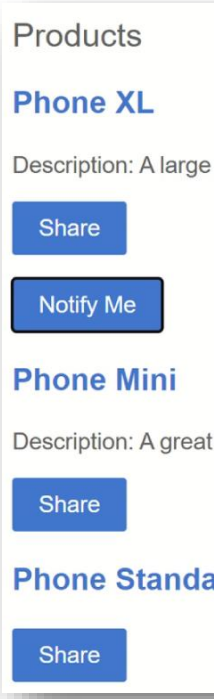


Strukturális direktíva - *ngFor

```
<div *ngFor="let product of products">

  <h3>
    <a
      [title]="product.name + ' details'"
      [routerLink]="['/products', product.id]">
      {{ product.name }}
    </a>
  </h3>

  <p *ngIf="product.description">
    Description: {{ product.description }}
  </p>

  <button type="button" (click)="share()">
    Share
  </button>

  <app-product-alerts
    [product]="product"
    (notify)="onNotify()">
  </app-product-alerts>

</div>
```

termékeken megyünk végig

HA van leírás, megjelenítjük




```
@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {

  products = products;

  share() {
    window.alert('The product has been shared
  }
```

Komponensek közti kommunikáció

- Komponensek (és direktívák) `@Input` és `@Output` paraméterekkel kommunikálhatnak egymással az adatkötésen keresztül
- Használhatnak megosztott szolgáltatásokat

```
<app-product-alerts
  [product]="product"
  (notify)="onNotify()">
</app-product-alerts>
```

```
onNotify() {
  window.alert('You will be notifie
}
```

```
@Component({
  selector: 'app-product-alerts',
  templateUrl: './product-alerts.component.html',
  styleUrls: ['./product-alerts.component.css']
})
export class ProductAlertsComponent {
  @Input() product: Product | undefined;
  @Output() notify = new EventEmitter();
}
```

```
<p *ngIf="product && product.price > 700">
  <button type="button" (click)="notify.emit()">Notify Me</button>
</p>
```

*ngFor: több elem generálása

- “let **x** of **c**” szintaktika

```
<li *ngFor="let n of numbers">{{n}}</li>
```

  - > **x** a ciklusváltozó
  - > **c** a gyűjtemény

- Megszerezhetjük az indexet is

```
<li *ngFor="let n of numbers; let i=index">{{i+1}}: {{n}}</li>
```

Lista változásának nyilvántartása

- Ha a lista elemeket azonosítani tudja az Angular, akkor nem kell mindig a teljes listát újra generálnia
- Egy metódust készítünk, ami visszaad egy kulcsot az elemhez
- Ha nem változik a kulcs, akkor nem generálja újra a hozzá tartozó elemet

```
trackByProducts(index: number, product: Product): number { return product.id; }
```

```
<div *ngFor="let product of products; trackBy: trackByProducts">
```

Egy strukturális direktíva elemenként

- Egy HTML elemre egy strukturális direktívát lehet tenni!
- Ha a **for** ciklus futtatását feltételhez szeretnénk kötni, akkor kívül legyen az **NgIf** és belül az **NgFor**
- Megjegyzés: az **<ng-container>** elem nem generál extra DOM objektumot, itt jól jön

Komponens ≥ direktíva

- A komponens egy speciális direktíva
  - Van saját template-je, amit kirajzol
  - Frissíti a template-ben levő adatkötést
  - Kezeli a felületről érkező eseményeket
- Az alkalmazás komponensek és direktívák hierarchiájaként épül föl

```
@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
```

Alkalmazás-életciklus

- Az angular alkalmazás = komponensek, direktívák életciklusát a keretrendszer vezérli
- A DI segítségével a sablonok feldolgozása után példányosítja a komponenseket, azok függőségeit
  - Később aszinkron események hatására új komponenseket, függőségeket gyárt

Komponens életciklusa

1. Példányosítás
   - Konstruktor hívása, amikor a keretrendszer észleli a DOM-ban
2. Renderelés
   - HTML készítése a sablon kiértékelése alapján
3. Változás követés
   - Bemenő, adatkötött értékek változásának automatikus figyelése
   - Változás esetén újra renderelés
4. DOM-ból eltűnve megszűnik a komponens

Lifecycle hooks

- Az életciklus eseményekre metódusok implementálásával "iratkozunk fel"
  - A metódusokhoz interfészek tartoznak, érdemes az interfészeket explicit implementálni
  - Metódus: ngOnInit
  - -> Interfész: OnInit

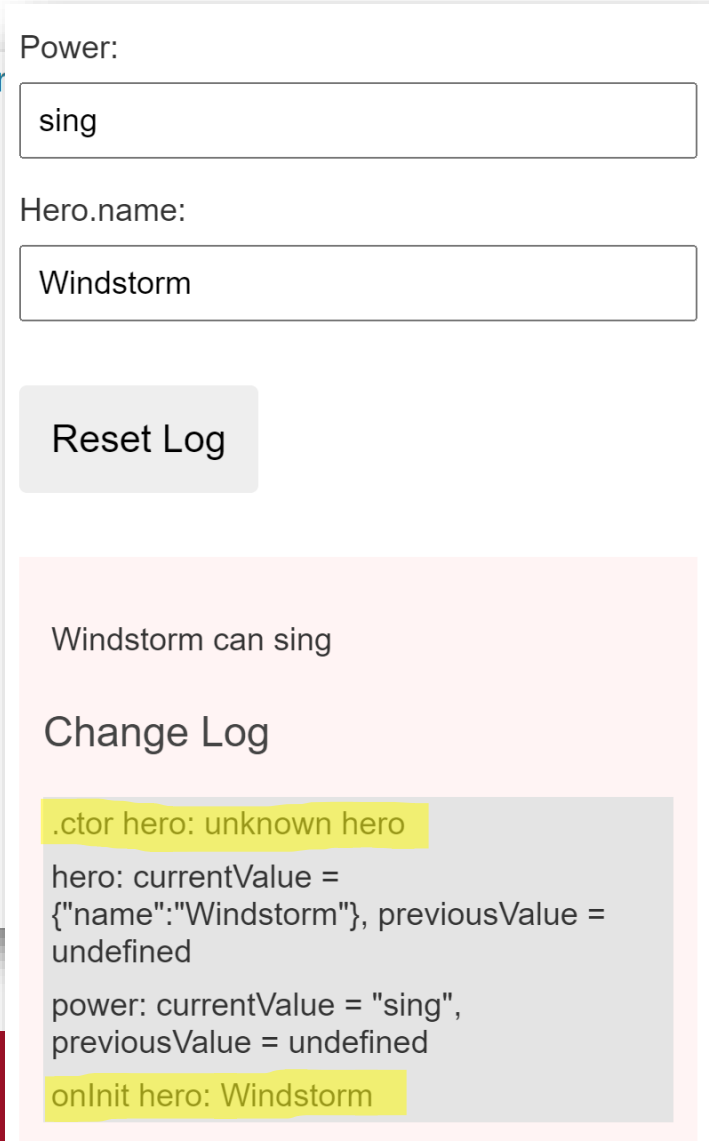
ngOnInit

- Egyszer hívódik meg, miután az adatkötés lefutott =@Inputtal ellátott változók beállítodtak
  - Előtte lefuthat az ngOnChanges, ha van adatkötés
- Mindig érdemes ebbe tenni az inicializációs logikát, mert már lefutott az adatkötés
  - Akkor is, ha a komponensben még nincsenek adatkötött változók, később lehet, hogy lesznek
- A konstruktor nyelvi elem, az objektum létrehozásakor fut le

.ctor vs OnInit

- A hero komponens beállítása

```
export class OnChangesComponent implements OnInit, OnChar
  @Input() hero!: Hero;
  @Input() power = '';

  changeLog: string[] = [];

  constructor()
  {
    this.changeLog.push(`.ctor hero:
          ${this.hero?.name ?? "unknown hero"}`);
  }

  ngOnInit(): void {
    this.changeLog.push(`onInit hero:
          ${this.hero?.name ?? "unknown hero"}`);
  }
```



ngOnChanges

- Mindig lefut, amikor változik a bemenő adat
  - Ha nincs adatkötés, nem fut le
- Gyakran történik, teljesítmény kritikus!
- A SimpleChange objektumban kapjuk meg a változásokat:
  - Változott property
  - Régi és új érték
  - Első változás-e?
- Nem figyeli a teljes objektumot, csak a kötött propertyt

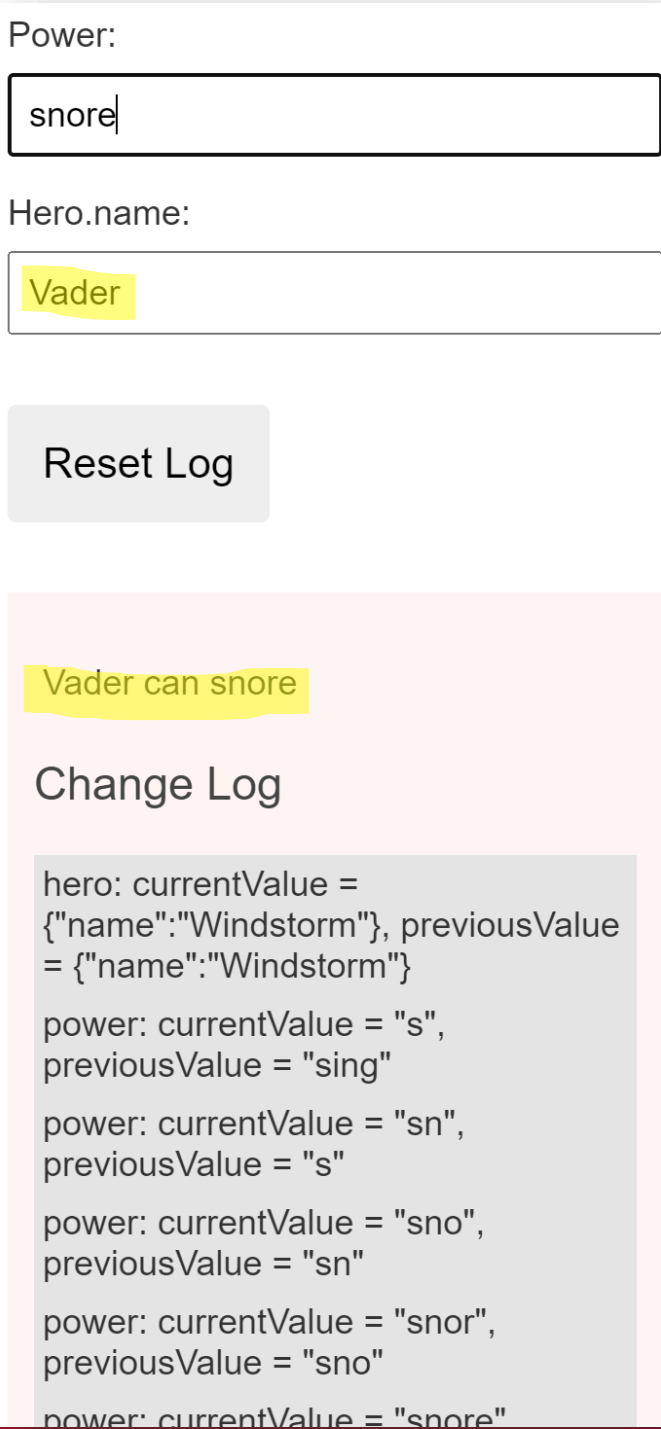
SimpleChange

- A name változásra nem fut le!

```
<div class="info">
  <p>{{hero.name}} can {{power}}</p>

  <h3>Change Log</h3>
  <div *ngFor="let chg of changeLog" class="log">{{chg}}</div>
</div>
```

```
export class OnChangesComponent
  @Input() hero!: Hero;
  @Input() power = '';
```

```
ngOnChanges(changes: SimpleChanges) {
  for (const propName in changes) {
    const chng = changes[propName];
    const cur  = JSON.stringify(chng.currentValue);
    const prev = JSON.stringify(chng.previousValue);
    this.changeLog.push(
      `${propName}: currentValue = ${cur},
                    previousValue = ${prev}`);
  }
}
```



NgDoCheck

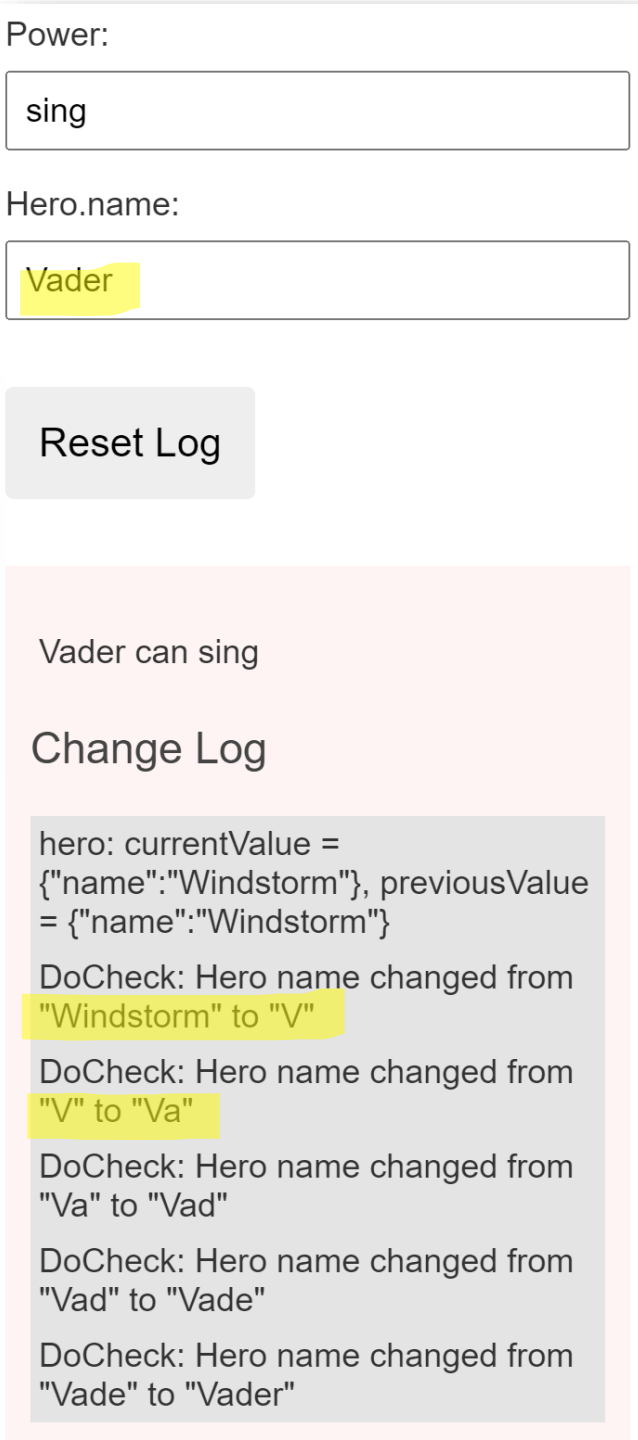
- Tetszőleges egyéb változás manuális észlelése
  - > Egyéni logika írható
- Az oldalon minden változásra lefut!
  - > Például máshova kerül a caret
- Nagyon gyakran hívódik -> teljesítmény kritikus!

Változások észlelése

- Minden változásra lefut!

```
ngDoCheck() {
  if (this.hero.name !== this.oldHeroName) {
    this.changeLog.push(`DoCheck: Hero name changed
        from "${this.oldHeroName}"
        to "${this.hero.name}"`);
    this.oldHeroName = this.hero.name;
  }
```

```
@Input() hero!: Hero;
@Input() power = '';
oldHeroName: string = "";
```



További események

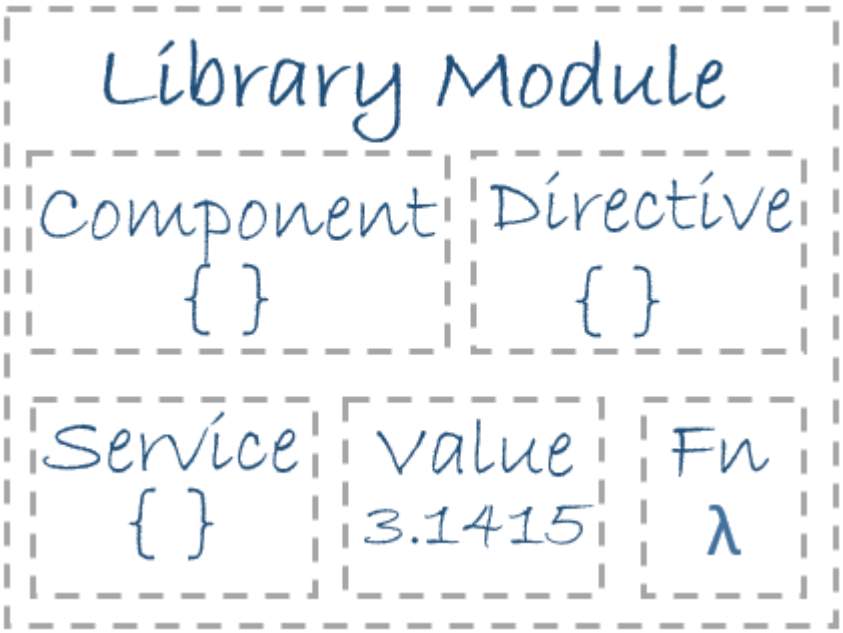
- A tartalom és nézet inicializációjáról és változást követő vizsgálatairól, valamint a komponens/direktíva megszűnéséről
- A DOM műveleteket érdemes a nézet inicializálásakor elvégezni
  - Kerüljük a kézi DOM műveleteket (ElementRef szolgáltatás), inkább használjunk adatkötést
  - Ahol nem elkerülhető, használjuk a Renderer2 szolgáltatást
  - Végső esetben nyúljunk csak a DOM-hoz közvetlenül

Angular Module / NgModule

- Egy alkalmazás önmagában összefüggő része
  - Komponensek, direktívák, …
  - Újrafelhasználás és/vagy modularizálás
- A keretrendszer maga is moduláris, mindig csak azokat a modulokat használjuk, amire szükségünk is van:
  - Core – mindig kell
  - Common – nagyon gyakori
  - Forms – űrlapok készítése
  - Http – http kommunikáció
  - Router - útvonalválasztás
  - …

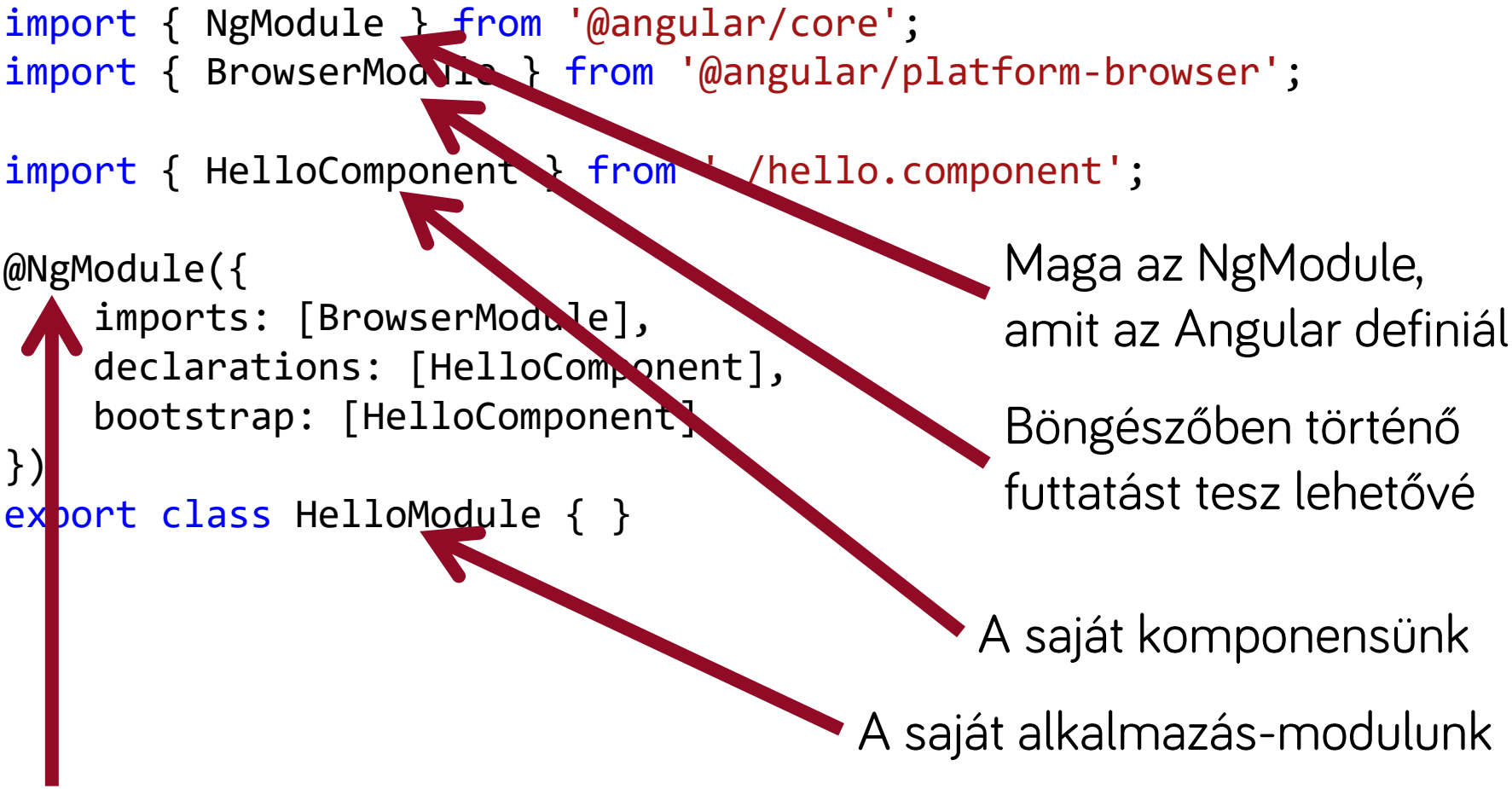
Angular osztálykönyvtárak

- A külső Angular alapú függőségeket modulokba szervezik, ezeket a saját modulunk függőségként definiálja



Hello Angular alkalmazás - NgModule

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

import { HelloComponent } from './hello.component';

@NgModule({
    imports: [BrowserModule],
    declarations: [HelloComponent],
    bootstrap: [HelloComponent]
})
export class HelloModule { }
```



A dekorátorral metaadatokat kötünk a modulhoz: függőségek, deklarációk, belépési pont

Hello Angular – bootstrapping (1)

- Az alkalmazás indításához szükséges

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

platformBrowserDynamic().bootstrapModule(HelloModule);
```

A HelloModule által definiált
alkalmazás indítása a böngészőben

Hello Angular – bootstrapping (2)

```
<!DOCTYPE html>
<html>
  <head>
    <!-- ... -->
  </head>

  <body>
    <hello-component>Loading...</hello-component>
  </body>
</html>
```

Az Angular indulásáig ez egy sima HTML elem, ezért a Loading... felirat jelenik meg a felületen

Hello Angular – bootstrapping (3)

1. Az Angular függőségeinek betöltése
   1. Polyfill a böngészőkompatibilitás miatt
   2. ZoneJS a végrehajtási környezetek szeparációjához
   3. RxJS aszinkron eseménykezeléshez
2. Alkalmazás belépési pontjának betöltése és indítása
3. Az Angular átveszi az alkalmazásindítási folyamatot

Hello Angular!

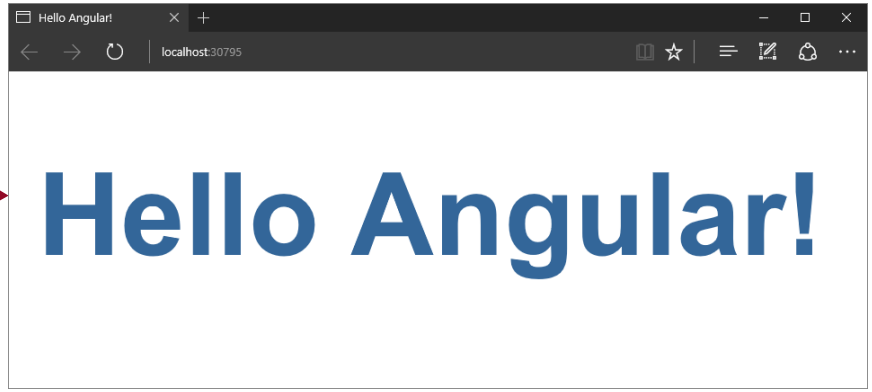
```
<body>
  <hello-component>Loading...</hello-component>
</body>
```

1. Az alkalmazás indulása után a bootstrappelt komponens illesztése

```
@Component({
  selector: 'hello-component',
  template: `<h1>Hello {{title}}!</h1>`
})
export class HelloComponent {
  title = 'Angular';
}
```

2. A kezdeti adatkötések végrehajtása

3. Renderelés a DOM-ba



Csővezetékek

- A pipe (csővezeték) a felületen megjelenő adat formázására szolgál
- Néhány beépített pipe lehet segítségünkre:
  - DatePipe
  - UpperCasePipe/LowerCasePipe
  - CurrencyPipe
  - PercentPipe
  - AsyncPipe

Paraméterezett, kapcsolt pipe

- Csővezeték paraméterezése

```
{{ birthday | date:'fullDate' }}
```

- Csővezetékek egymás után láncolása

```
{{ birthday | date:'fullDate' | uppercase }}
```

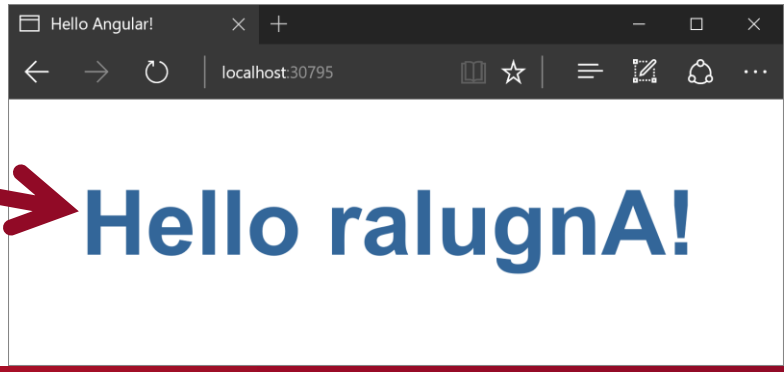
ReversePipe – példa

Angular @Pipe dekorátor és megnevezés

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

@Pipe({ name: 'reverse' })
export class ReversePipe implements PipeTransform {
  transform(value: string): string {
    return value.split("").reverse().join("");
  }
}
```

Bármilyen transzformációs függvény

```
<h1>Hello {{title | reverse}}!</h1>
```



Rendezés, szűrés pipe-okkal

- Rendezési, szűrési csővezetéket **nem** ad az Angular, mert nagyon rossz teljesítményhez vezet
  - Érdemes csak az alkalmazás logikájában szűrnünk és rendeznünk
  - A keretrendszer figyeli az adatkötött objektumok változásait, hogy szükség esetén újrarajzolja a felületet

Pure vs impure pipe

- A „tiszta” pipe nem detektálja a változásokat a megadott modellben, kivéve, ha a modellje:
  - Egyszerű típus (String, Boolean, Number, Symbol)
  - Referencia típus (Object, Function, Date, Array…) esetén **csak akkor**, ha a referencia változik!
- Ha mindig vizsgálódnunk kell, referencia típus változása esetén, impure pipe-ot használunk: az ilyen „tisztátalan” pipe-ot jeleznünk kell metaadatban is

```
@Pipe({ name: 'filter', pure: false })
```

  - Alapértelmezetten tiszta, az összehasonlítás gyors – az impure pipe esetén viszont teljes bejárás történik minden felhasználói és egyéb aszinkron eseményre
    - Teljesítménykritikus

Köszönöm a figyelmet!

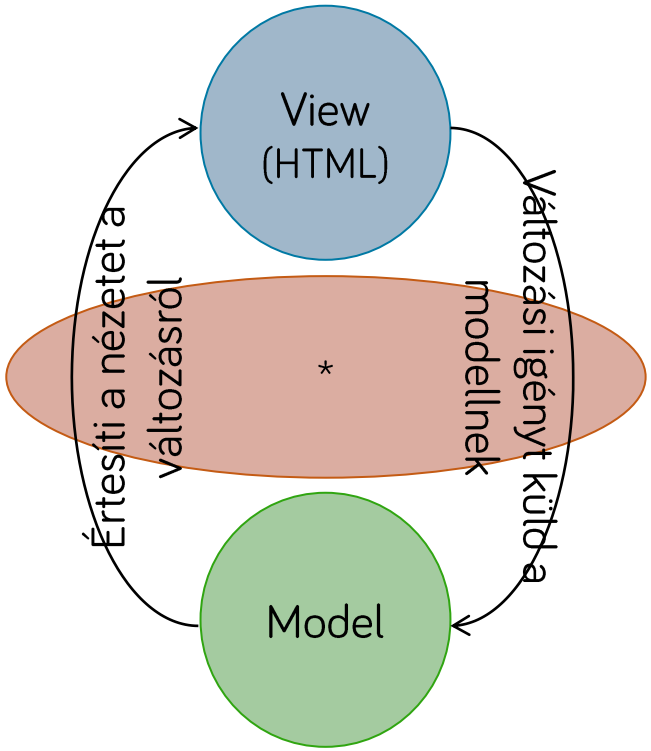
Angular

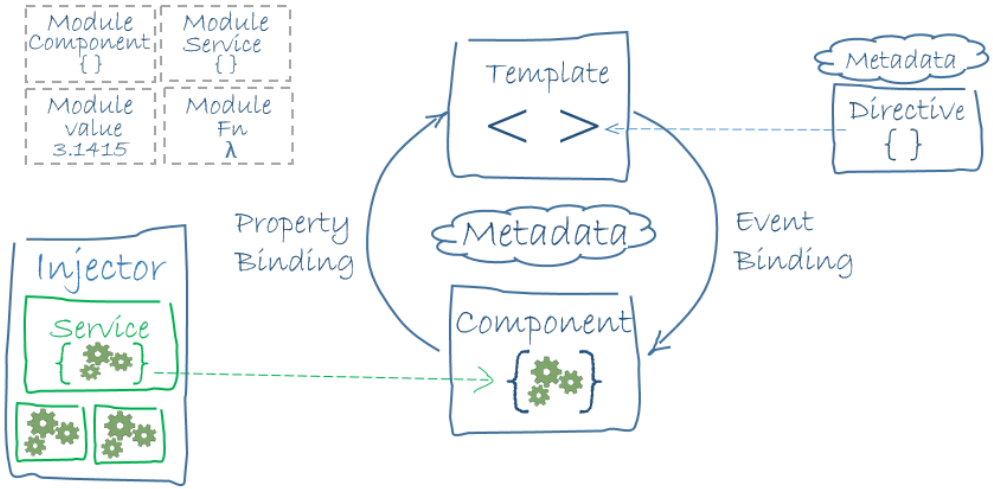
Második előadás

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Architektúra

- MV*, adatkötés, dependency injection, single responsibility, separation of concerns





Metaadatok

- TypeScript dekorátorok
  - @NgModule, @Component, @Directive, @Input, @Output, @Inject, @Injectable

```
@Component({
  selector: 'hello-component',
  template: `<h1>Hello {{title}}!</h1>`
})

@NgModule({
    imports: [BrowserModule],
    declarations: [HelloComponent],
    bootstrap: [HelloComponent]
})

@Directive({ selector: '[myHighlight]' })
```

Komponensek

- Funkcionális egység saját adatkötést kezelő kóddal (@Input, @Output) és sablonnal, ami a felületen megjelenik
- Hivatkozhat más komponensekre a template-ben -> komponens-hierarchia

```
import { Component } from '@angular/core';
@Component({
  selector: 'hello-component',
  template: `<app-greet>Hello {{title}}!</app-greet>`
})
export class MyComponent {
  title = 'Angular';
}
```

Tartalom

Szolgáltatások

- Az üzleti logikát, felületfüggetlen módon szolgáltatásokba tudjuk szervezni
- A szolgáltatások jól meghatározott háttérfunkciót látnak el vagy infrastrukturális funkciót adnak: HTTP kommunikáció, adattárolás/-visszanyerés, naplózás, felhasználókezelés
  - > Vagy jól leválasztható üzleti logikai funkciókat tesznek elérhetővé

```
export class RandomDataService {
  constructor() {}
  getData() {
    return [1, 2, 5, 6, 3];
  }
}
```

Szinkronitás és számosság

	Szinkron	Aszinkron
Egy érték	Függvényhívás	Promise
Több érték	Enumeráció	Observable

24

Routing

- A routing (útvonalválasztás) feladata, hogy ad
  URL-t megfelelő komponens-hierarchiához és
  alkalmazásállapot-halmazhoz rendelejen
- A RouterModule definiálja (`@angular/router`)
  - > Direktívák: RouterOutlet, RouterLink, RouterLinkA
  - > Konfiguráció: Routes
- Lehetséges hierarchikus útvonalválasztást is megvalósítani
- Alapértelmezetten a HTML5 history API-t használja

29

Űrlapok

- Az űrlapok egységes adatbeviteli felületek
- Gyakori feladatok üzleti alkalmazásokban:
  - > Adatbekérő/-módosító felület
  - > Felületi validáció, hibaüzenet megjelenítése
- Két alapvető űrlap-készítési módszer létezik:
  - > Sablon alapú (template-driven)
    - Egyszerűbb, direktívákra épít, aszinkron
  - > Modell alapú (reactive)
    - Komplexebb, tesztelhető, jobban újrafelhasználható, szinkron

35

Komponens stílusa

- Klasszikus CSS-sel (vagy SCSS, Sass, Less) stilizálható
- A stílusokat a komponens hivatkozza be
  - > Inline vagy .css fájlt hivatkozva
- Minden komponens **saját stílusokkal** rendelkezik, a többitől függetlenül
  - > Ezzel biztosítva az újrafelhasználást
- A globális stílusok érintik a komponens megjelenését

53

Animáció

- Az animáció a CSS tranzíciós funkciój
  - > Pozíció, méretek, transzformáció, szín, ke
- Importálni kell az animációs modult é
  használni a komponensekben

```
import { BrowserAnimationsModule } from '@angular/platform-browser/
@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule
  ],
  declarations: [ ],
  bootstrap: [ ]
})
export class AppModule { }
```

```
import { Component, HostBinding }
import {
  trigger,
  state,
  style,
  animate,
  transition,
  // ...
} from '@angular/animations';
```

Szolgáltatások

- Az üzleti logikát, felületfüggetlen módon szolgáltatásokba tudjuk szervezni
- A szolgáltatások jól meghatározott háttérfunkciót látnak el vagy infrastrukturális funkciót adnak: HTTP kommunikáció, adattárolás/-visszanyerés, naplózás, felhasználókezelés
  - > Vagy jól leválasztható üzleti logikai funkciókat tesznek elérhetővé

```
export class RandomDataService {
  constructor() {}
  getData() {
    return [1, 2, 5, 6, 3];
  }
}
```

Szolgáltatások (service)

- Tipikusan nem globálisan hozzuk létre a komponensen kívüli objektumokat
  - > Szolgáltatások formájában
  - > Nem kódoljuk bele egyik komponensbe sem
  - > Felsoroljuk, hogy mik vannak, mi hivatkozik mire
    - -> **deklaratív** megközelítés
- A keretrendszer automatikusan létrehozza
- -> **Függőség injektálás** (Dependency Injection)

Függőségek injektálása

```
export class DiceComponent implements OnInit {

  rand: RandomDataService;

  constructor() {
    this.rand = new RandomDataService();
  }
```

```
export class DiceComponent implements OnInit {

  rand: RandomDataService;

  constructor(rand:RandomDataService) {
    this.rand = rand;
  }
```

Függőség injektálás (DI)

- Általánosan: A komponens által használt függőséget nem a komponens kezeli, hanem csak kapja. Az injektor felelős a függőségek kezeléséért (életciklus, kiosztás)
  - A komponens nem felelős a szolgáltatások **létrehozás**áért, használatra kapja őket
  - Nem kell ismernie a **függőségi gráf**ot
  - A komponensek jönnek-mennek, míg a szolgáltatások **életciklusa** lehet teljesen eltérő

```
constructor(rand:RandomDataService) {
  this.rand = rand;
}
```

Függőség injektálás (DI)

- Szerepkörök szétválasztása
  - Nő a tesztelhetőség
  - Erősödik a dekompozíció
  - Javul az újrafelhasználhatóság
  - Komponens kódja kisebb, olvashatóbb marad
- Nem csak Angularban van
  - Így ismerhetjük fel a DI-t: használunk egy szolgáltatást (külső funkcionalitást), ami nem globális és nem is mi hoztuk létre

Példa szolgáltatás

- Attól szolgáltatás, hogy injektálható

```
import { Injectable } from '@angular/core';
@Injectable({ providedIn: 'root' })
export class RandomDataService {
  constructor() {}
  getData() {
    return [1, 2, 5, 6, 3];
  }
}
```

- Amúgy csak egy sima osztály
- providedIn: mely komponensek számára elérhető
  - > root mindenkinek

Szolgáltatás felhasználása

- Átvesszük a konstruktorban, és használjuk
- Az injektor gondoskodik arról, hogy
  - Létre legyen hozva, saját függőségeivel együtt
  - Átadja, amikor létrejön a komponens
  - Megszüntesse valamikor

```
rand: RandomDataService;
data: number [];

constructor(rand:RandomDataService) {
  this.rand = rand;
  this.data = [];
}

ngOnInit(): void {
  this.data = this.rand.getData();
}
```

DI – példa

Injektálás engedélyezése az osztályon

```
@Injectable()
export class Logger {
  // capture logs for testing
  logs: string[] = [];
  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

```
import { Logger } from './services/logger';

@NgModule({
  providers: [Logger],
```

Bejegyzés a modulba

```
import { Logger } from '../services/logger';

@Component({
  selector: 'app-hello',
  template: `<h1 (click)="log()">
              Hello {{title}}!</h1>`,
})
export class HelloComponent {
  title: string = 'Angular';

  constructor(public logger: Logger) {}

  log() {
    this.logger.log(this.title);
  }
}
```

Konstruktor-injektálás

Függőség szkópja

- Ki fér hozzá a függőséghez, szolgáltatáshoz?
  - > Attól függ, hol regisztráljuk a providert
- Globális
  - > Egyetlen singleton példány jön létre
- Modul szintű
  - > A modulon belül érhető el egyetlen példány
- Komponens szintű
  - > A komponens és a sablonjában használt további komponensek, direktívák számára
  - > Új szolgáltatás példány minden új komponens példányhoz

```
@Injectable({ providedIn: 'root' })
export class Logger {
```

```
@NgModule({
  providers: [Logger],
```

```
@Component({
  providers: [Logger],
```

Példányosítás testreszabása

- A providerben megadott érték valójában egy kulcs és nem osztály, ami azonosítja a létrehozandó függőséget
- Állítható a létrehozás módja
  - **useClass**: ezt a konkrét osztályt használja
  - **useExisting**: meglévő regisztráció
  - **useValue**: előre létrehozott globális objektum
  - **useFactory**: külön factoryt adunk meg

```
@Component({
  providers: [Logger],
```

```
@Component({
  providers: [{ provide: Logger, useClass: Logger }],
```

```
@Component({
  providers: [{ provide: Logger, useClass: ConsoleLogger }],
```

Interfész DI-ban

- Interfész nem használható kulcsként, mert nincs futás idejű reprezentációja JavaScriptben

```
@Component({
  providers: [{ provide: ILogger, useClass: ConsoleLogger }],
```

```
'ILogger' only refers to a type, but is being used as a value here. (2693)
```

- Megoldások:
  - > Interfész helyett **absztrakt osztály**
  - > A kulcs legyen sztring → törékeny = runtime error

```
@Component({
  providers: [{ provide: "ILogger", useClass: ConsoleLogger }],
```

```
constructor(@Inject("ILogger") public logger: ILogger) {}
```

Manuális DI kulcs létrehozása

- Explicit hozzuk létre a kulcsot
  - A típusosságon sajnos nem segít ☹

```
▶ ERROR TypeError: this.logger.log is not a function
    at HelloComponent.log (main.js:325:21)
```

```
export const LOGGER = new InjectionToken<ILogger>('logger');
```

```
import { ConsoleLogger, Logger, LOGGER, ILogger } from '../services/logger';

@Component({
  providers: [{ provide: LOGGER, useClass: ConsoleLogger }],
```

```
constructor(@Inject(LOGGER) public logger: ILogger) {}
```

Szolgáltatások

- Szolgáltatások
  - > HttpClient: XHR+json kommunikáció
  - > Location: address bar
  - > FormBuilder: űrlap kezelő
  - > Router: navigáció
  - > …
- A tree-shaking működik globális szolgáltatásokra
  - > Csak azok kerülnek bele a végső kódba, amire hivatkozunk

HttpClient

- A HttpModule-ban található (`@angular/common/http`)
  - Nem feltétlenül kommunikál az appunk HTTP-n, ezért külön modul
- A HttpClient szolgáltatás injektálásával XHR (AJAX) kommunikációt folytathatunk
- **Ne** a komponenseinkben, hanem a szolgáltatásainkban használjuk (separation of concerns)
- Alapértelmezetten **JSON** alapú **HTTP** kommunikációt végez

HttpClient – példa szolgáltatás

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/http';
import { Observable } from "rxjs/Rx";

import { Product } from './product';

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

  getProducts() : Observable<Product[]> {
    return this.http.get<Product[]>("/api/products");
  }
}
```

HttpClient szolgáltatás injektálása

Aszinkron működés Observable használatával

Adat lekérése GET-tel URL-ről

HttpClient – példa komponens

```
import { Component, OnInit } from "@angular/core";
import { Product } from "./product";
import { ProductService } from "./product.service";

@Component({
  selector: "product",
  template: `<div *ngFor='let product in products'>
               {{product.name}} <b>{{product.price}}</b>
             </div>`
})
export class ProductComponent implements OnInit {
  products: Product[];
  constructor(private productService: ProductService) { }
  ngOnInit(): void {
    this.productService.getProducts()
      .subscribe(products => this.products = products);
  }
}
```

Adathalmaz bejárása (amint elérhetővé válik)

Adat eltárolása tagváltozóban

HttpClient – példa válasz megfigyelése

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/http';
import { Observable } from "rxjs/Rx";

import { Product } from './product';

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

  getProducts() : Observable<HttpResponse<Product[]>> {
    return this.http.get<Product[]>("/api/products",
                                    observe: 'response');
  }
}
```

Az observe paraméter megadásával nem csak az adatot, hanem a teljes HTTP választ elkérhetjük (pl. fejlécek, státuszkód)

Szinkronitás és számosság

	Szinkron	Aszinkron
Egy érték	Függvényhívás	Promise
Több érték	Enumeráció	Observable

Observable

- Az **Observable** egy tervezési minta, mely szerint időben érkező események egy *adatfolyamot* (stream) írnak le, amire reagálhatunk
  - A *subject* értesíti az *observereket* az állapota megváltozásairól
- Az Observable-ök:
  - Transzformálhatók, szűrhetők
  - Több kezelő iratkoztatható fel rájuk
  - Az RxJS keretrendszer megvalósítja
- Az egyszer bekövetkező Observable visszavezethető egyszerű Promise-ra

Aszinkronitás + Angular + Observable (1)

product.component.html

```
<div id="search-component">
  <h4>Product Search</h4>
  <input #searchBox (keyup)="search(searchBox.value)" />

  <div>
    <div *ngFor="let product of products | async"
         (click)="gotoDetail(product)" class="search-result" >
      {{product.name}} <b>{{product.price}}</b>
    </div>
  </div>
</div>
```

Template-ben nevesített DOM elem

Eseménykezelő billentyű felengedésére

async pipe az Observable kiburkolására

Aszinkronitás + Angular + Observable (2)

```
@Component({
  selector: "product",
  templateUrl: "product.component.html"
})
export class ProductComponent implements OnInit {
  products: Observable<Product[]>;
  private searchTerms = new Subject<string>();

  constructor(private productService: ProductService) { }

  search(term: string): void {
    this.searchTerms.next(term);
  }
  ngOnInit(): void {
    this.products = this.searchTerms.asObservable()
      .pipe(switchMap(term => this.productService.search(term));
  }
}
```

Template külső fájlban

Az adat az Observable-ből jön majd

Saját Observable alany

A termékek a kulcsszavak projekciója HTTP kéréssé

Aszinkronitás + Angular + Observable (3)

```
ngOnInit(): void {
    this.products = this.searchTerms
      .pipe(debounceTime(500))

      .pipe(distinctUntilChanged())

      .pipe(switchMap(term => term
        ? this.productService.search(term)
        : Observable.of<Product[]>([])))

      .catch(error => {
        console.log(error);
        return Observable.of<Product[]>([]);
      });
  }
```

500ms-enként csak az utolsó eseményt engedjük át

Ha nem változik a kifejezés értéke, nem vesszük új eseménynek

Egy új Observable-t gyártunk feltételtől függően: ha a term üres, akkor üres tömb a válasz

Routing

- A routing (útvonalválasztás) feladata, hogy adott URL-t megfelelő komponens-hierarchiához és alkalmazásállapot-halmazhoz rendelejen
- A RouterModule definiálja (`@angular/router`)
  - Direktívák: RouterOutlet, RouterLink, RouterLinkActive
  - Konfiguráció: Routes
- Lehetséges hierarchikus útvonalválasztást is megvalósítani
- Alapértelmezetten a HTML5 history API-t használja

Routing konfiguráció

Factory metódus a helyben konfigurált RouteModule regisztrációjához

```
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      { path: 'products', component: ProductsListComponent },
      { path: 'search', component: SearchComponent },
      { path: 'products/:id', component: ProductDetailsComponent }])
  ],
  declarations: [AppComponent, ProductsListComponent,
                 SearchComponent, ProductDetailsComponent],
  bootstrap: [AppComponent]
})
export class AppModule { title = 'products'; }
```

Komponensek útvonalhoz rendelése

URL paraméter, amit a komponens az ActivatedRoute szolgáltatás segítségével ki tud nyerni

RouterOutlet, RouterLink

- RouterOutlet: ez a direktíva végzi maguknak a komponenseknek a példányosítását a megadott helyen a DOM-ban
- RouterLink: ez a direktíva egy horgony (<a>) elemhez a megfelelő linket és működést rendeli
- RouterLinkActive: a link aktivitásának megfelelően a megadott CSS classt rendeli az adott DOM elemhez

Linkek

A két főbb útvonalat a felhasználó a linkekkel, vagy az URL megadásával választja ki

app.component.html

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/products" routerLinkActive="active">
    Products
  </a>
  <a routerLink="/search" routerLinkActive="active">
    Search products
  </a>
</nav>
<router-outlet></router-outlet>
```

A stílusozást testreszabjuk az `"active"` CSS class segítségével

A Routerbe regisztrált komponensek közül az aktuális a RouterOutlet helyén jelenik meg

Feltételes navigáció: Guard

- A navigáció megakadályozható, ha bizonyos feltételek nem teljesülnek
- Ehhez egy Guard példányt regisztráltunk a DI-ba és az adott útvonalhoz rendeljük

Feltételes navigáció: Guard

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
import { LoginService } from './login.service';

@Injectable()
export class AuthGuardService implements CanActivate {
    constructor(private loginService: LoginService) { }
    canActivate(): boolean {
        return !!this.loginService.user;
    }
}

[
    { path: 'login', component: LoginComponent },
    { path: 'profile', component: ProfileComponent,
                       canActivate: [AuthGuardService] }
    ...
]
```

Guard interfész implementációja

Guarddal védett útvonal

Űrlapok

- Az űrlapok egységes adatbeviteli felületek
- Gyakori feladatok üzleti alkalmazásokban:
  - Adatbekérő/-módosító felület
  - Felületi validáció, hibaüzenet megjelenítése
- Két alapvető űrlap-készítési módszer létezik:
  - Sablon alapú (template-driven)
    - Egyszerűbb, direktívákra épít, aszinkron
  - Modell alapú (reactive)
    - Komplexebb, tesztelhető, jobban újrafelhasználható, szinkron

Sablon alapú formok

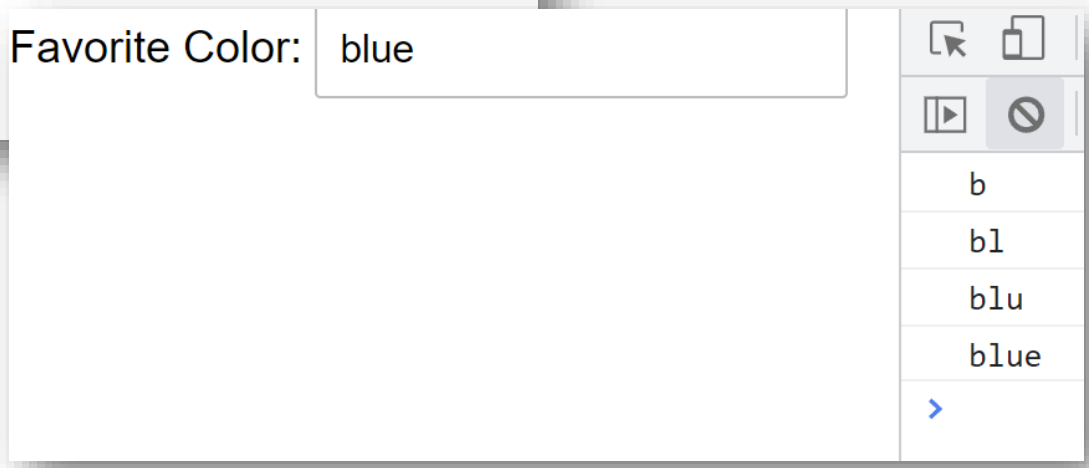
- A form **vezérlők kezelése implicit**, nincs közvetlen hozzáférésünk a FormControlokhoz
- Mindezt az **NgModel** direktíva intézi

```
@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class TemplateFavoriteColorComponent {
  favoriteColor = '';
}
```

Adatfolyam

- Minden változás megjelenik a modellünkben

```
@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class TemplateFavoriteColorComponent {
  _favoriteColor = '';
  get favoriteColor() { return this._favoriteColor; }
  set favoriteColor(color: string) {
    console.log(color);
    this._favoriteColor = color; }
}
```



Form

- **Két irányú adatkötés**: **változások** követése a form vezérlői és a mögöttes **adatmodell** között
- Fontsabb osztályok
  - **FormControl**: egy form vezérlő változás követése, validációs állapota
  - **FormGroup**: több vezérlőhöz kötött értékek és állapot nyomonkövetése
  - **FormArray**: dinamikus űrlap támogatás
  - **ControlValueAccessor**: a híd az Angular FormControl példány és a beépített DOM elem között

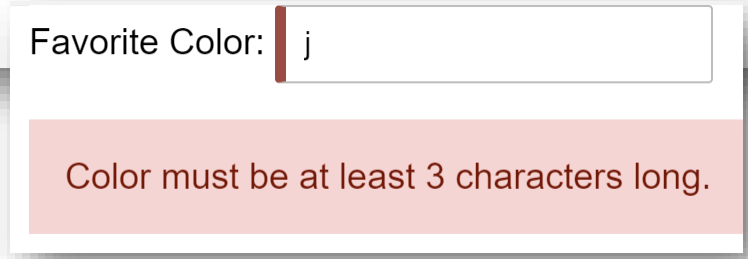
Sablon alapú formok ellenőrzése

- **HTML alapú validáció**
- Amikor változik egy vezérlő értéke, lefut az ellenőrzés -> az ngModel-en elérhető az eredmény
- A #...-kal készíthetünk egy **sablon változót** az ngModel alapján, ami kapcsolódik az alatta lévő FormControlhoz
  - **status**: valid / invalid / pending / disabled
  - **valid**: boolean, megfelelő-e az érték
  - **dirty**: változtatta-e az értéket a felhasználó
  - **touched**: elhagyta-e a mezőt a felhasználó
  - **errors**: validációs hibák tömbje

Sablon form ellenőrzés példa

- Az ellenőrzés a HTML-be van beleírva

```
Favorite Color: <input type="text" [(ngModel)]="favoriteColor"
  #favColor="ngModel"
  minlength="3" required>

<div *ngIf="favColor.invalid && (favColor.dirty || favColor.touched)"
  class="alert">

<div *ngIf="favColor.errors?.['required']">
  A color is required.
</div>
<div *ngIf="favColor.errors?.['minlength']">
  Color must be at least {{favColor.errors?.['minlength'].requiredLength}} characters long.
</div>
</div>
```



Reaktív formok

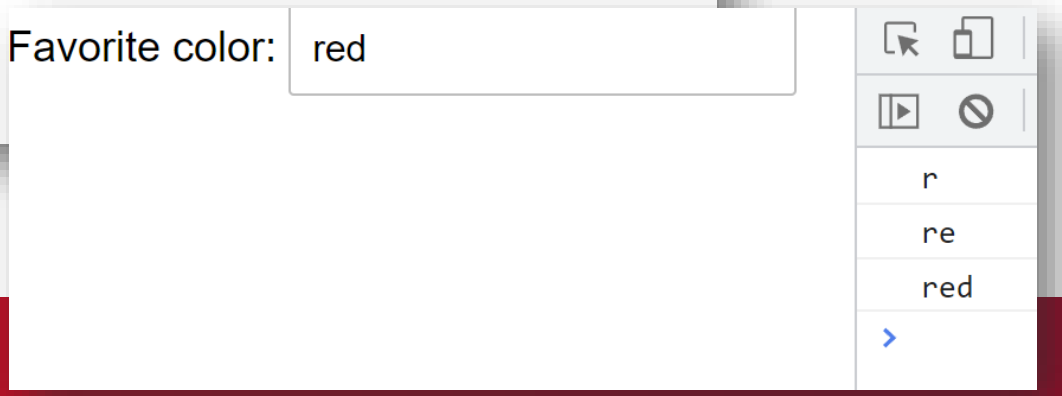
- **Közvetlen hozzáférés a objektum modellhez**
  - A **formControl** direktíva köti hozzá a HTML elemet a vezérlő objektumhoz

```
@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text" [formControl]="favoriteColorControl">
  `
})
export class ReactiveFavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

Szinkron adatfolyam

- A változásokat közvetlenül a form controlokon érzékeljük

```
@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite color: <input type="text" [formControl]="favoriteColorControl">
  `
})
export class ReactiveFavoriteColorComponent implements OnInit {
  favoriteColorControl = new FormControl('');

  ngOnInit() {
    this.favoriteColorControl.valueChanges.subscribe(v=>console.log(v))
  }

}
```

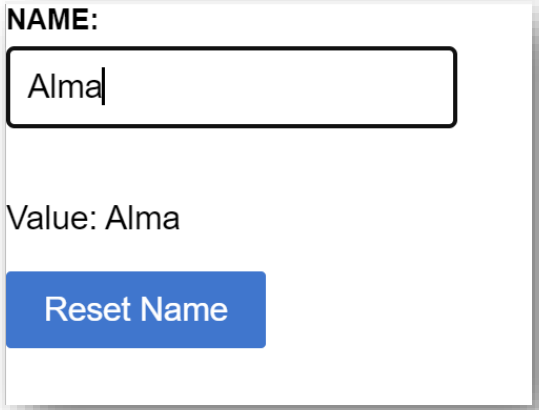


A vezérlő programozottan elérhető

- Lekérdezés: **.value**
- Beállítás: **setValue**

```ts
@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl('');

  resetName() {
    this.name.setValue('');
  }
}
```

```html
<label for="name">Name: </label>
<input id="name" type="text" [formControl]="name">

<p>Value: {{ name.value }}</p>

<button type="button" (click)="resetName()">Reset Name</button>
```



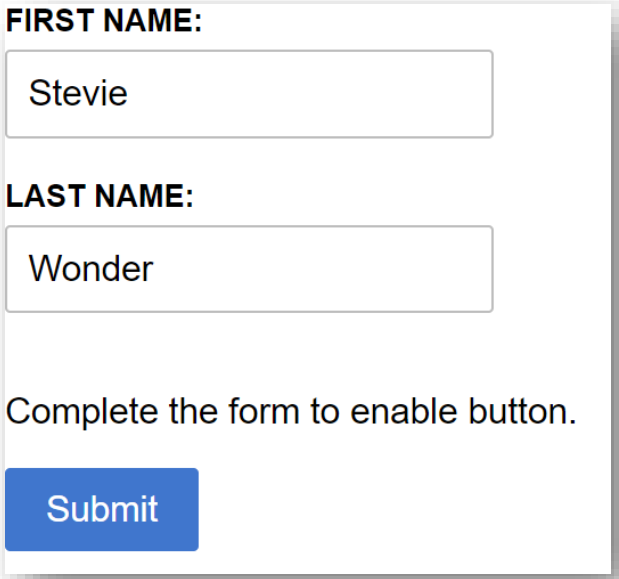
Több vezérlő összefogása

- **FormGroup**: vezérlők fix halmaza
- **FormArray**: dinamikusan alakítható futás időben
- Egymásba ágyazhatók, így a formok darabolhatók külön részekre
  - Mindkettő az **AbstractControl**ból származik, ahogy a FormControl is

Egyszerű form példa submittal

```ts
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
  });

  onSubmit() {
    // TODO: Use EventEmitter with form value
    console.warn(this.profileForm.value);
  }
}
```

```
{firstName: 'Stevie', lastName: 'Wonder'}
  firstName: "Stevie"
  lastName: "Wonder"
```




```html
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

<label for="first-name">First Name: </label>
<input id="first-name" type="text" formControlName="firstName">

<label for="last-name">Last Name: </label>
<input id="last-name" type="text" formControlName="lastName">

<p>Complete the form to enable button.</p>
<button type="submit" [disabled]="!profileForm.valid">Submit</button>
</form>
```

Form érték beállítása

- A **setValue** mindig a teljes objektumot várja, aminek a struktúrája (strukturális típusosság) egyezik a form alakjával
- A **patchValue**-nak elég az objektum egy részét átadni, csak azt fogja módosítani

```
updateProfile() {
  this.profileForm.patchValue({
    firstName: 'Stevie',
  });
```

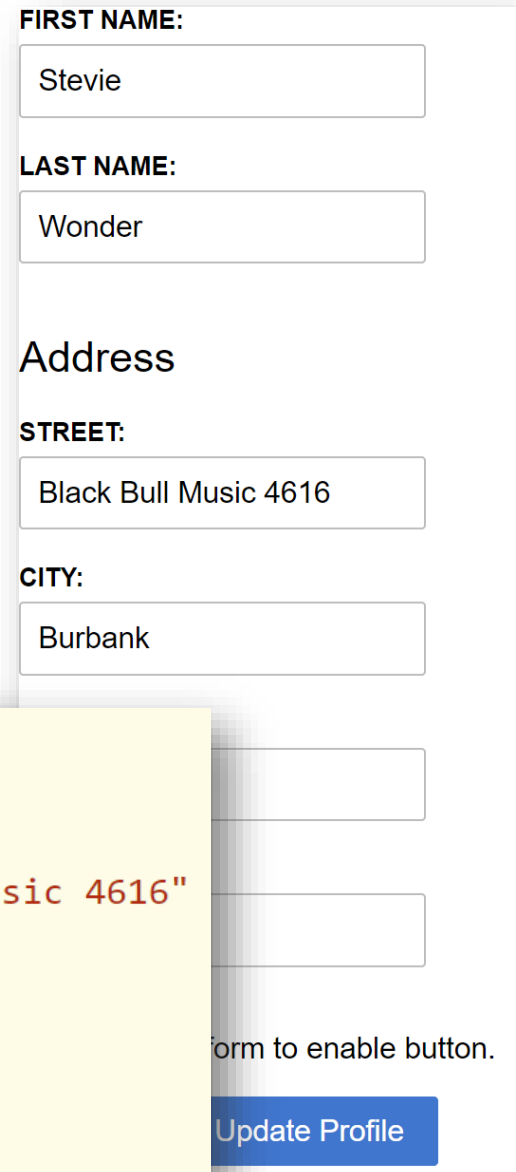
Control generálás

- **FormBuilder** segítségével tömörebb kód

```
export class ProfileEditorComponent {
  constructor(private fb: FormBuilder) { }

  profileForm = this.fb.group({
    firstName: [''],
    lastName: [''],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });
```

```
▼address:
    city: "Burbank"
    state: "CA"
    street: "Black Bull Music 4616"
    zip: ""
  ▶[[Prototype]]: Object
  firstName: "Stevie"
  lastName: "Wonder"
▶[[Prototype]]: Object
```

```
<div formGroupName="address">
<h2>Address</h2>

<label for="street">Street: </label>
<input id="street" type="text" formControlName="street">

<label for="city">City: </label>
```



Reaktív form ellenőrzése

- Beépített vagy saját ellenőrző kód a **komponensben**
- A form objektum **status/...** propertyjei jelzik az állapotot

```
profileForm = this.fb.group({
  firstName: ['',
  [Validators.required,
    Validators.minLength(3)]],
  lastName: [''],
  address: this.fb.group({
    street: [''],
    city: [''],
    state: [''],
    zip: ['']
  }),
});
```

```
get firstName() : AbstractControl
{ return this.profileForm.get('firstName')!; }
```

```
<input id="first-name" type="text" formControlName="firstName">
<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touc
    class="alert alert-danger">
  <div *ngIf="firstName.errors?.['required']">
    First name is required.
  </div>
  <div *ngIf="firstName.errors?.['minlength']">
    First name must be at least {{firstName.errors?.['minlength'].r
  </div>
</div>
```

```
<p>Complete the form to enable button.</p>
<button type="submit" [disabled]="!profileForm.valid">
```

```
<p>Form Status: {{ profileForm.status }}</p>
```

FIRST NAME:

Al

First name must be at least 3 characters long.

Complete the form to enable button.

Submit

Form Status: INVALID

Komponens stílusa

- Klasszikus CSS-sel (vagy SCSS, Sass, Less) stilizálható
- A stílusokat a komponens hivatkozza be
  - Inline vagy .css fájlt hivatkozva
- Minden komponens **saját stílus**okkal rendelkezik, a többitől függetlenül
  - Ezzel biztosítva az újrafelhasználást
- A globális stílusok érintik a komponens megjelenését

Komponens stílusok példa

- A komponens stílusok csak belül hatnak
- A globális stílusok érintik a komponenst is

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

```
<button class="red-button">Button</button>
```

```
.red-button {
  background: red;
}
```

```
<app-root></app-root>
<button class="red-button">Button</button>
```

Button Button

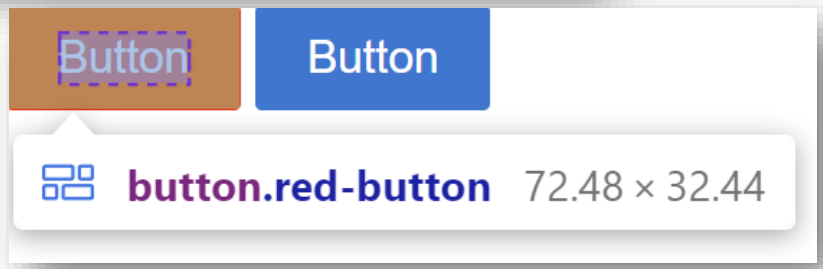
```
/* Button */
.button, button {
  display: inline-flex;
  align-items: center;
  padding: 8px 16px;
  border-radius: 2px;
  font-size: 14px;
  cursor: pointer;
  background-color: #1976d2;
  color: white;
  border: none;
}
```

A színfalak mögött...

- Minden stilizált elem a komponensben el van látva külön propertyvel
  - > A komponens host elemen még külön property van
- A stílusok így specifikusak lesznek

```
▼<app-root _nghost-xib-c48 ng-version="13.3.5">
    <button _ngcontent-xib-c48 class="red-button">Button</button> flex
     == $0
  </app-root>
  <button class="red-button">Button</button> flex
```

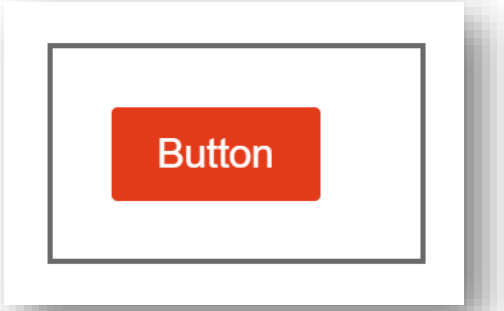
```
.red-button[_ngcontent-xib-c48] {
   background: ▶ red;
}
```



A host elem stilizálása

- A speciális :host selectorral magát a komponenst hosztoló elemet tudjuk stilizálni, nem a tartalmat

```
:host {
  border: 2px solid dimgray;
  display: block;
  padding: 20px;
  margin: 20px
}
```



Témák

- A **:host-context**-tel feltételesen érvényesíthetünk stílusokat, ha a szülő hierarchiában megjelenik a hivatkozott CSS osztály – tipikusan témázásra használjuk

Témázható komponens

```
<button class="btn-theme">Button</button>
```

```
:host-context( .light-theme ) .btn-theme {
  background-color: #f1e4aa ;
  color: #161616 ;
}

:host-context( .dark-theme ) .btn-theme {
  background-color: #161616 ;
  color: #f1e4aa ;
}
```

Téma beállítása a szülőn

```
<div class="dark-theme">
  <app-theme-button></app-theme-button>
</div>

<div class="light-theme">
  <app-theme-button></app-theme-button>
</div>
```

Animáció

- Az animáció a CSS tranzíciós funkciójára épül
  - > Pozíció, méretek, transzformáció, szín, keret stb.
- Importálni kell az animációs modult és aztán használni a komponensekben

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';


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

```
import { Component, HostBinding } from '@angular/core';
import {
  trigger,
  state,
  style,
  animate,
  transition,
  // ...
} from '@angular/animations';
```

Animáció beállítása

- A stílushoz és HTML sablonhoz hasonlóan **deklaratívan** tudjuk beállítani a különböző **állapotok** megjelenését és az **átmenet** módját

```
@Component({
  selector: 'app-my-button',
  templateUrl: './my-button.component.html',
  styleUrls: ['./my-button.component.css'],
  animations: [
    // ...
  ]
})
export class MyButtonComponent implements OnInit {
```

Állapotok leírása

név + megjelenés

```
state(
  'open',

  style({
    height: '200px',
    opacity: 1,
    backgroundColor: 'yellow',
    color: 'black',
  })
),
```

The box is now Open!

```
state(
  'closed',

  style({
    height: '100px',
    opacity: 0.8,
    backgroundColor: 'blue',
    color: 'lightgray',
  })
),
```

The box is now Closed!

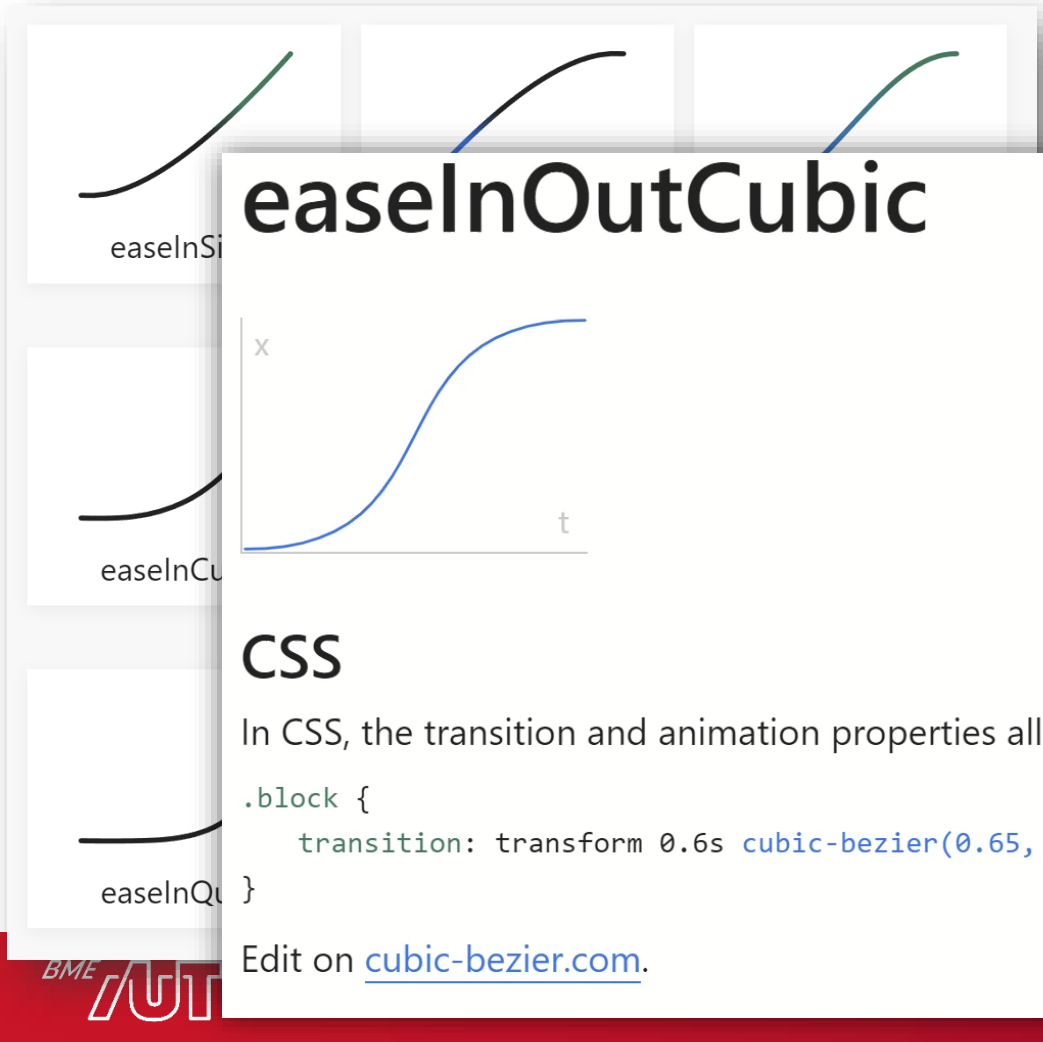
Változás leírása

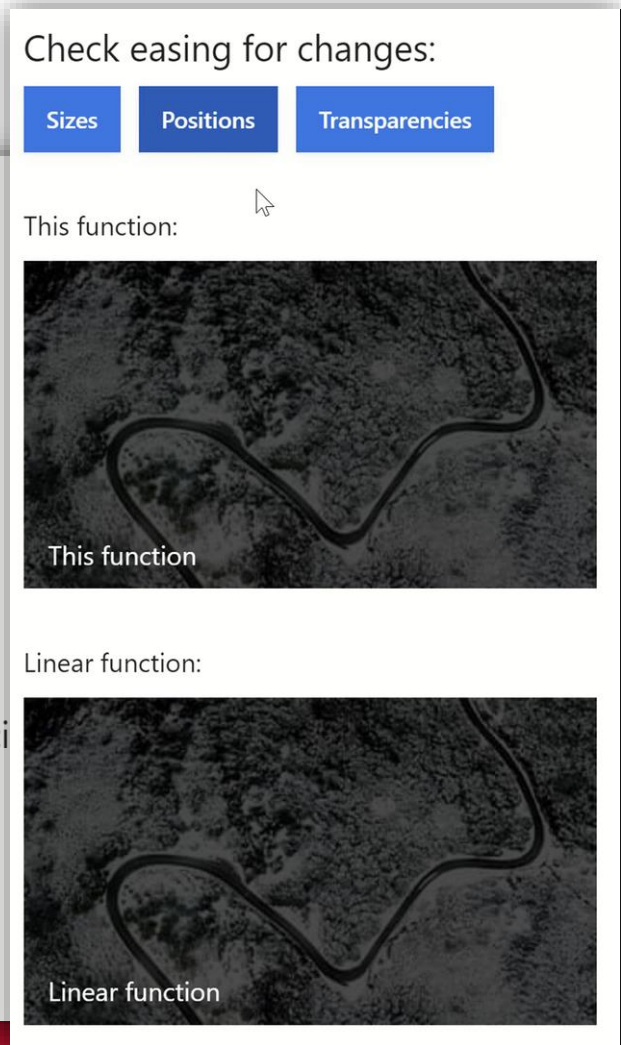
- Az **animate** funkcióval, paraméterei:
  - duration - időtartam: 100 / '100ms' / '0.1s'
  - [delay] – késleltetés
  - [easing] – lefutás
- **Lefutás**
  - Ha nem adjuk meg, akkor lineáris ☹
  - ease, ease-in, ease-out, ease-in-out
  - cubic-bezier(...)

```
[animate('1s ease-out')]

[animate('0.5s cubic-bezier(0.4, 0.0, 0.2, 1)')]
```

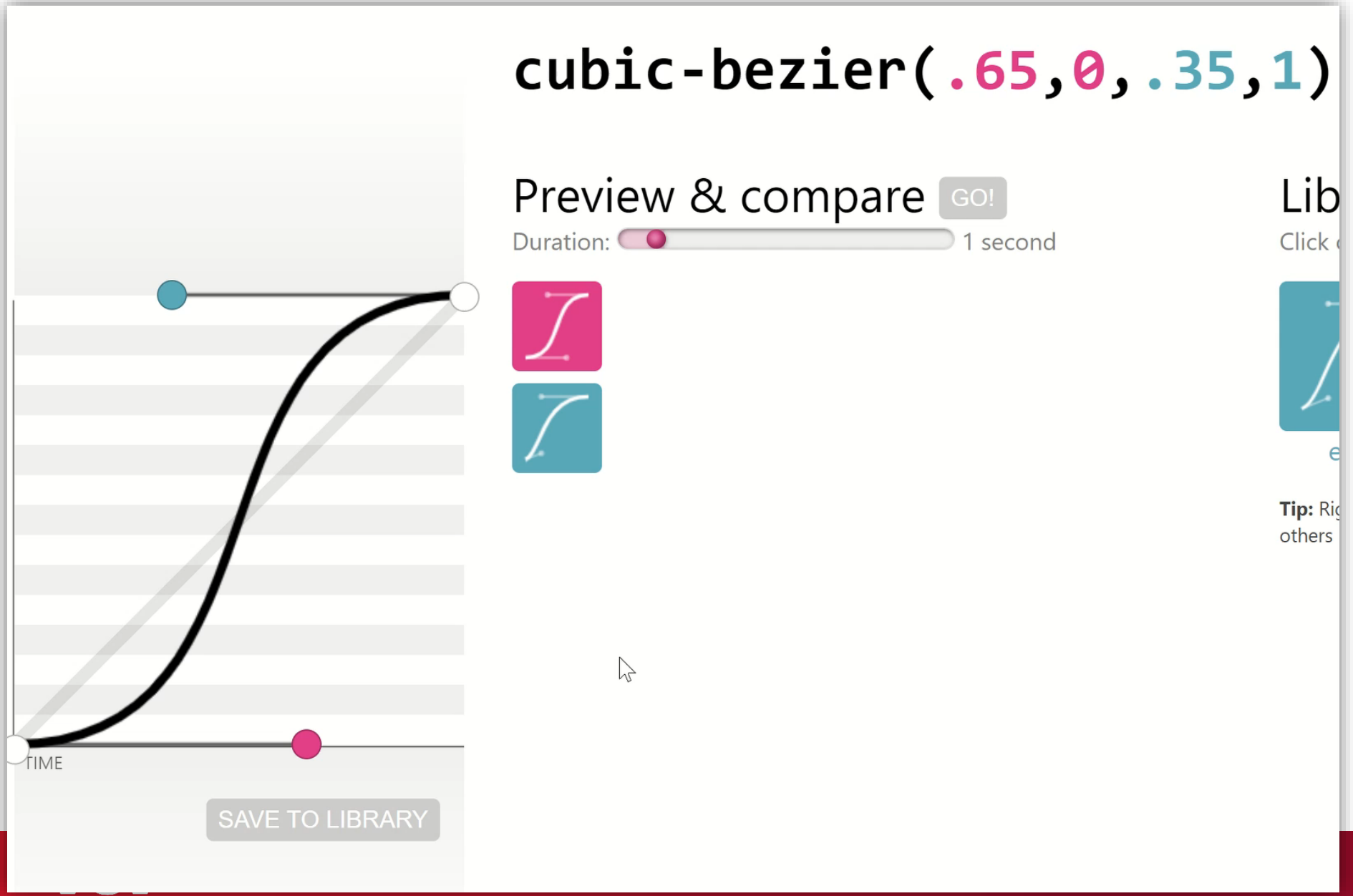
easings.net

- Több lefutás közül választhatunk, előnézettel





cubic-bezier.com/



material.io/design/motion/speed.html#easing

- Material dizájn nyelv vonatkozó része
- Javaslatot ad különböző helyzetekre

Platform	Protocol
Android	FastOutSlowInInterpolator
CSS	cubic-bezier(0.4, 0.0, 0.2, 1);
Flutter	standardEasing
iOS	[[CAMediaTimingFunction alloc] i

Átmenetek beállítása

- Megadjuk, milyen állapotok között milyen átmenetet használjon
- **transition** funkció:
  - Állapot név
    - *'open => closed'*
  - Mindegyik állapot: *
    - '* => *'
    - 'open => *' : openből bármelyik állapotba
  - Eltűnés / megjelenés: void
    - '* => void' : eltűnés bármilyen állapotból

```
transition('open => closed', [
  animate('1s')
]),
```

Animáció indítása: trigger

- Az animációnak **nevet adunk**
- **trigger** funkció:

  állapotok + **változások** + **átmenetek** leírása

```
animations: [
  trigger('openClose', [
    state( 'open', style({···
    ),
    state( 'closed', style({···
    ),
    transition('open => closed', [animate('1s ease-out')]                          ),
    transition('closed => open', [animate('0.5s cubic-bezier(0.4, 0.0, 0.2, 1)')]  ),
  ]),
],
```

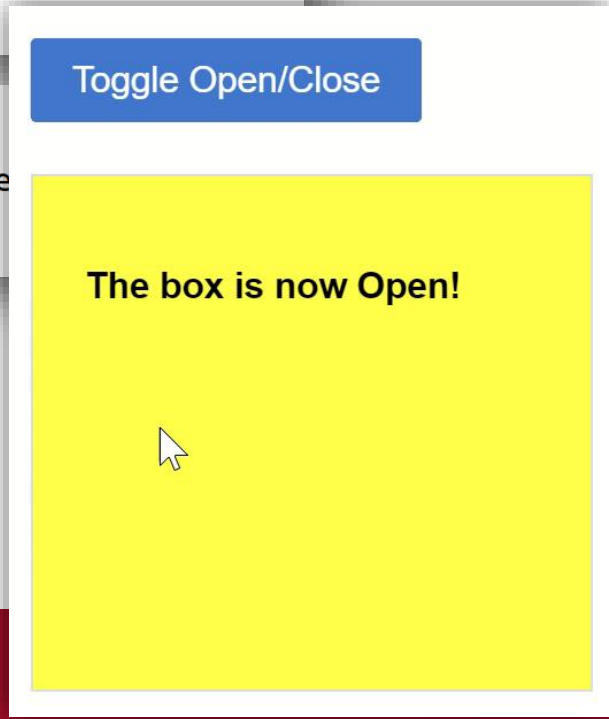
Animáció HTML-hez kötése

- @ jellel hivatkozzuk a triggert a komponensen belül
- Megadjuk a **cél állapotot**

```
<div [@openClose]="isOpen ? 'open' : 'closed'" class="open-close-container">
  <p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>
</div>
```

```
<nav>
  <button type="button" (click)="toggle()">Toggle Open/Close</button>
</nav>
```

```
export class OpenCloseComponent {
  isOpen = true;

  toggle() {
    this.isOpen = !this.isOpen;
  }
}
```



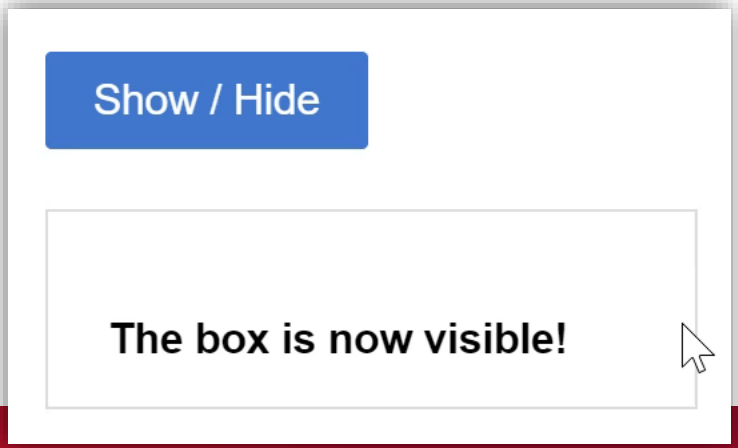
Megjelenés és eltűnés

- Az :enter és :leave rövidítéseket használhatjuk
  - > Hasznos a *ngIf és *ngForral együtt

```
<nav>
<button type="button" (click)="toggleVisibility()">Show / Hide</button>
</nav>
```

```
<div *ngIf="isVisible" @fadeSlideInOut class="open-close-container">
<p>The box is now {{ isVisible ? 'visible' : 'hidden' }}!</p>
</div>
```

```
trigger('fadeSlideInOut', [
  transition(':enter', [
    style({ opacity: 0,
      transform: 'translateY(10px)' }),
    animate('500ms', style({ opacity: 1,
      transform: 'translateY(0)' })),
  ]),
  transition(':leave', [
    animate('500ms', style({ opacity: 0,
      transform: 'translateY(10px)' })),
  ]),
```



További animációs lehetőségek

- Több elem animálása: **query** funkcióval a szülőn
- Elemek animálása kis késleltetéssel: **stagger**
- Párhuzamosan futó animációk egy elem különböző propertyjeire: **group**
- Route váltásnál
- **Újrafelhasználható**, paraméterezett animációk
- Debuggoláshoz kikapcsolás: NoopAnimationsModule
- https://indepth.dev/posts/1285/in-depth-guide-into-animations-in-angular

Mire érdemes odafigyelni?

- Dizájn
  - **Ne legyen zavaró!** Megterhelő lehet!
  - Adjon hozzá a vizuális élményhez
  - A nagyon **kis mozgások** igényessé tehetik a kinézetet
  - Használjunk **organikus** változásokat!
- Élmény = User eXperience
  - Segítsen a felhasználónak **követni az eseményeket**
    - Például egy elem törlésénél: mi történik a képernyőn, melyik elem tűnik el?
  - Az észlelést irányítva vezesse át a felhasználót az **alkalmazás egy másik állapotába**, vizuálisan