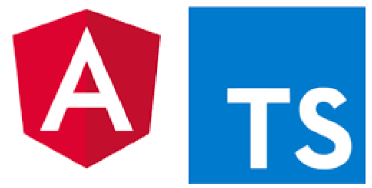
Univerzitet u Beogradu – Elektrotehnički fakultet

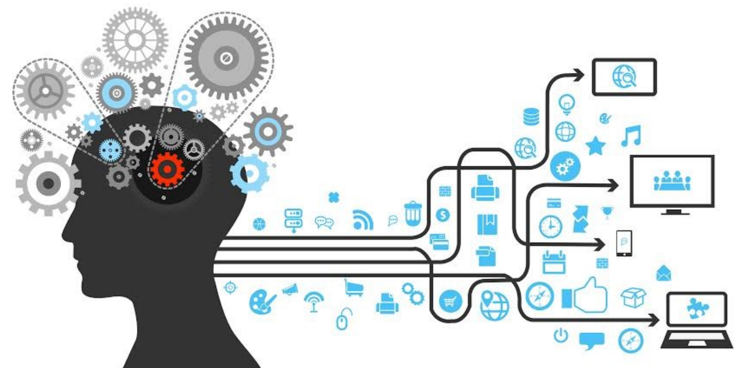
Uvod u Angular i TypeScript

Predavač: Dražen Drašković

Beograd, 28. mart 2024. godine

Sadržaj





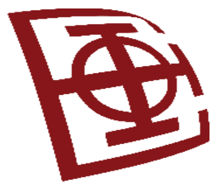
- Uvod
- Kratak pregled veb aplikacija
- JavaScript programski jezik
- Angular JS (Angular 1.x)
- Angular 2+
- TypeScript programski jezik
- Veb aplikacije sa jednom stranom (*Single Page App*)
- MEAN i MERN arhitekture
- JS radni okviri kao alternative

Uvod

- Glavni akteri veb aplikacija:
  korisnik, veb pregledač, veb server i pozadinski servisi
- Korisnici: ostvaruju interakciju i imaju očekivanja
- Veb pregledač: mora biti podržan od strane tehnologije,
  jer on komunicira od veb servera i ka veb serveru
- Komunikacija:
  - GET zahtev - najčešće za preuzimanje podataka;
  - POST zahtev - za slanje podataka na server;
  - AJAX zahtev – GET ili POST zahtev koji direktno šalje JS zahteve pregledaču.

Pregled programskih jezika i tehnologija kroz godine

1971-1980	1981-1990	1991-2000	2001-2010	2011-данас
1972: C	1981: MS-DOS1	1991: Python	2001: Win XP	2011: Android 4
1972: Prolog	1983: Ada 83	1991: Visual Basic	2003: Scala	2012: Win 8
1968: Algol68	1983: C ++	1991: DOS5, Linux	2003: Red Hat	2012: TypeScript
1974: SQL	1983: Turbo Pascal	1992: Open GL	2004: JSF 1.0	2013: An.JellyBean
1978: Matlab	1983: Word for DOS	1993: Solaris	2004: Gmail Mozilla Firefox	2013: An.KitKat
1979: Atari DOS	1984: Lisp	1995: Delphi	2005: Ubuntu 5	2013: React JS
1980: Ada 80	1985: Windows 1.0	1995: Java / Jscript	2007: MS Vista	2013: JSF 2.2
1980: TCP / IP	1985: NSF NET	1995: PHP, Ruby	2008: Android	2014: HTML 5
	1987: Perl	1995: Windows 95	2008: Goo.Chrome	2014: Hack (FB)
	1987: Windows 2.0	1996: Windows NT	2009: CoffeeScript	2014: Angular 2
	1987: Excel for Win	1997: Mac OS 7 / 8	2009: Win 7	2014: Swift
	1988: MS-DOS 4	1998: C++ stand. JavaServlet stand.	2009: Mongo DB	2016: Kotlin (2011)
	1989: WWW/HTML	1998: Windows 98	2009: Node JS	2017: C++17; JavaServlet 4
	1990: Windows 3.0	2000: C#	2010: Angular JS	2018: Angular 7



JavaScript / ECMAScript

- 1995. razvijen kao klijentski skript jezik
- HTML + CSS + JavaScript = jezgro WWW
- Strukturno programiranje preuzeto iz jezika C, a sličan Javi
- Jezik viših paradigmi, podržava: event-driven, funkcionalne, imperativne stilove (objektno-orijentisan / prototipski-baziran)
- Podržava rad sa: osnovnim tipovima podataka, tekstom i datumom, nizovima, regularnim izrazima, osnovne manipulacije sa DOM (*Document Object Model*)
- Ne podržava: rad sa IO, umrežavanje, ili grafičke objekte
- Danas ugrađen u mnoge softvere i serverski orijentisane tehnologije

Angular tehnologije

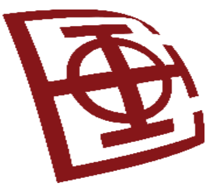
- Angular JS, oktobar 2010.
- Razvijen od strane Google kompanije i zajednice
- JavaScript baziran radni okvir za klijentsku stranu aplikacije
- Podržava razvoj aplikacija na jednoj veb strani (eng. *Single-page Web App*) i progresivnih veb aplikacija
- Uzorci: podržava MVC (i MVVM), što je dovelo do MVW
- JavaScript dopunjuje radni okvir Apache Cordova, koji služi za multiplatformi (*cross platform*) razvoj mobilnih aplikacija
- Link: https://angularjs.org/ (nema više podršku!)

Projektni uzorci kod današnjih aplikacija

Naziv	Komponente	Način povezivanja, prednost paradigme, i primer
MVC	Model View Controller	USER => Controller => Model => RDBMS => Model => View **+ Dobro razdvojena logika..** **Primer: Java, PHP, Ruby, C#**
MVP	Model View Presenter	*Presenter* menja *Controller* (*Presenter* 1-1 *View*, *Cont.* 1-many *View*) **+ Poboljšana testablinost.** **Primer: ASP.NET WebForms, Swing, Google WT**
MVVM	Model View ViewModel	*VM* komponenta = stanje podataka u modelu; (*VM* kao *P*, ali nema referencu na komp. *View*) **+ Odvaja GUI bolje od logike.** **Primer: MS WPF (.NET), Angular 6, EmberJS**
MVW	Model View Whatever	Whatever = { C, P, VM } **Primer: Angular JS**

AngularJS način rada (1)

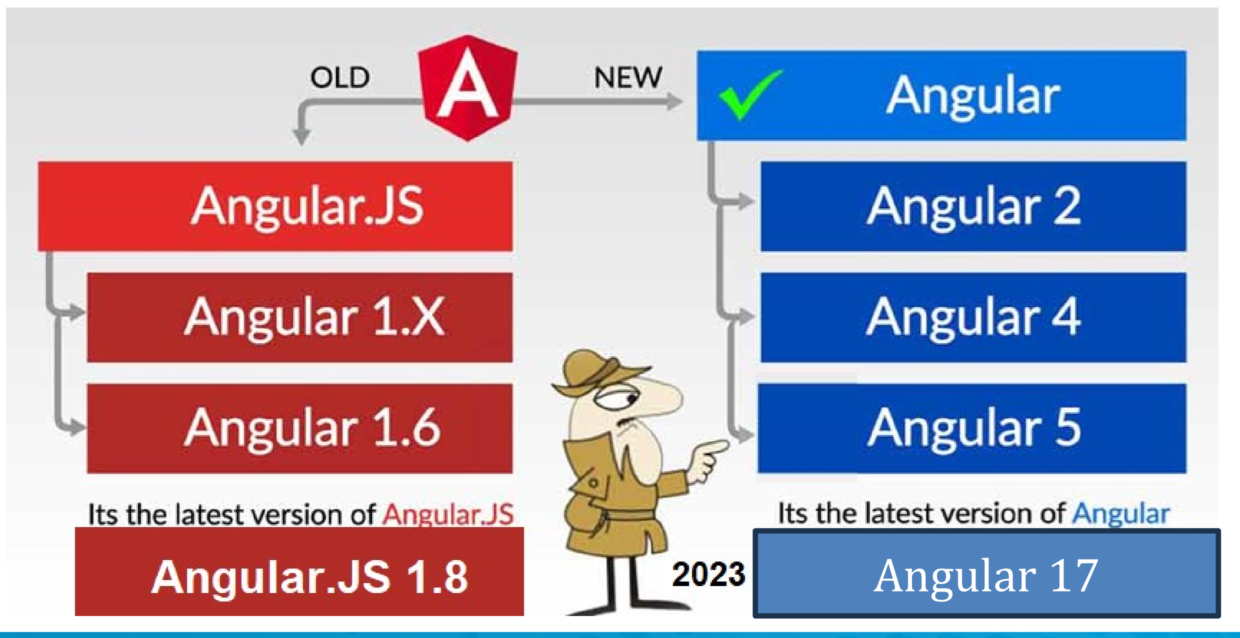
- AngularJS prvo učitava HTML stranu, koja ima ugrađene dodatne prilagođene tag atribute.
- Angular interpretira te atribute kao direktive za povezivanje ulaznih i izlaznih delova strane,
  sa modelom koji predstavlja standardne JavaScript varijable.
- Vrednost tih JS varijabli:
  manuelno unutar koda ili preuzeti iz statičkih/dinamičkih JSON resursa.

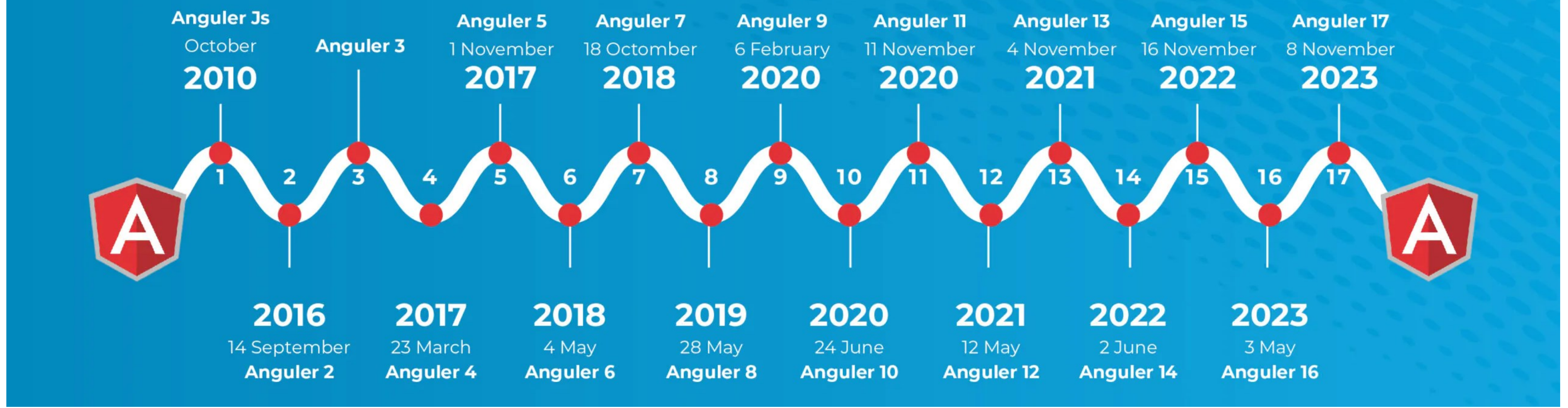


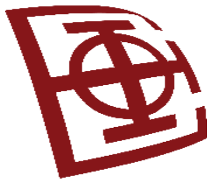
AngularJS način rada (2)

- Deklarativno programiranje, za korisničke intefejse i povezivane komponenti, pre imperativnog programiranja za definisanje poslovne logike.
- Obuhvata: DOM manipulaciju, validaciju ulaza, serversku komunikaciju, upravljanje URL,...
- Koristi MVC uzorak i dvosmerno povezivanje podataka (automatska sinhronizacija *Model*-a i *View*)
- Fokus na podršci za programiranje velikih i aplikacija sa jednom veb stranom (poboljšane performanse, testabilnost, reupotreba komponenti, moduli)

AngularJS => Angular 2+







AngularJS / Angular verzije

- Angular JS 1.0 (2010), Angular JS 1.2 (2012),
  Angular JS 1.3 (2014), Angular JS 1.6 (2016),
  Angular JS 1.7 (2018), Angular JS 1.8 (2020), 1.1.2022.-THE END
- 2014: Angular 2.0 (nekompatibilan sa AngularJS)

// preskočeno: Angular v3.3.0 (zbog neusaglašenosti)

- 2016: Angular 4.0 (kompatibilan sa Angular 2)
- 2017: Angular 5 (podrška za progresivne veb aplikacije, izgradnja optimizacija, i poboljšanja *Material Design*)
- 2018: Angular 6

…

- 2020: Angular 10
- **14. februar 2024: Angular 17.2.1**

Šta su ključne razlike u verzijama? (1)

- Arhitektura:
  - AngularJS: MVC / MVW
  - Angular 2: komponentna (servis/kontroler) arhitektura (ne postoji mogućnost nadogradnje iz AngularJS)
  - Angular 4: zasnovan na Angular 2, samo je smanjen dati programski kod i brži je razvoj
- Programski jezik:
  - AngularJS primenjuje JavaScript u razvoju
  - Angular 2 primenjuje TypeScript (superset programskog jezika JavaScript)
  - Angular 17.0.x kompatibilan sa TypeScript 5.2.0 ili višom verzijom

Šta su ključne razlike u verzijama? (2)

- Mobilni razvoj:
  - Angular 2 omogućio da se ostvari izvorna aplikacija za mobilne platforme (npr. *React Native*)
  - Angular 2 daje aplikacioni sloj i sloj prikazivanja (*rendering*)
  - Po potrebi, bilo koji pogled (*view*) može da se prikaže u realnom vremenu, za zahtevanu komponentu
- Komponente zasnovane na korisničkom interfejsu:
  - Koncept kontrolera u Angular JS potpuno eliminisan u Angular 2
  - Verzije 2+ svoj korisnički interfejs zasnivaju na komponenti
  - Verzije 2+ su poboljšale fleksibilnost i ponovnu upotrebu koda

TypeScript programski jezik

https://www.typescriptlang.org

JS vs TS vs ES

- *JavaScript* (JS)
  - programski jezik za izgradnju dinamičkih sajtova, razvoj na klijentskoj strani;
  - nisu potrebni nikakvi resursi na serveru; za aplikacije do 1000 linija koda;
  - koristi se sa drugim tehnologijama kao što su REST API, XML, i drugi.
- *TypeScript* (TS)
  - statički građen jezik za pisanje JavaScript koda, koristi se u tehnologijama Angular i Node.JS u savremenim veb aplikacijama;
  - superset jezika JS.
- *EcmaScript* (ES)
  - standard koji je imao za cilj da JS dovede u rang sa drugim programskim jezicima;
  - JS poboljšanje: ECMAScript 6 (preimenovan u ECMAScript 2015)
  - Chrome i Firefox su najpoznatiji veb pregledači koji podržavaju ES6.

JS vs TS

JavaScript	TypeScript
Ne podržava opcione parametre.	Podržava opcione parametre.
Interpreterski jezik koji označava greške u vremenu izvršavanja (*runtime*).	Kompajlira koda i označava greške tokom vremena razvoja (*compiletime*).
JavaScript nema podršku za module.	TypeScript ima podršku za module.
Brojevi i string su objekti.	Brojevi i string su interfejsi.
JavaScript ne podržava generike.	TypeScript podržava generike.

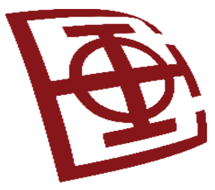
Tipovi podataka (1)

- String – znakovni tip, sa jednostrukim ili dvostrukim navodnicima
```
var mojTekst: string = 'PIA';
var drugiTekst: string = "PIA Novo";
```
- Broj – tip podataka za numeričke vrednosti
```
var broj: number = 10;
var cena: number = 133.55;
```
- Bulove vrednosti - tip podataka sa vrednostima true/false (kod flegova)
```
var da: boolean = true;
var ne: boolean = false;
```
- niz - indeksirani niz (indeksi od 0 do n-1 za n elemenata)
```
var niz: string[] = ["PIA","MIPS","TS"];
var prvi = niz[0];
var niz2:number[] = [1,2,3];
var niz3: Array<number> = [1,2,3,4,5];
```

Koristi se i ključna reč let umesto var: **let** godine: number = 30;

Tipovi podataka (2)

- torka - ako je poznat ustaljen broj elemenata niza
```
var x: [string, number];
x = ["Drazen", 20];
```
- enum - nabrojivi tip, može dodeljivati i određene vrednosti
```
enum Ljudi {Jelica, Sanja, Bosko, Drazen}
var x = Ljudi.Bosko; //number 2
var y = Ljudi[0]; //string Jelica
enum Boje {Crvena=1, Zelena, Plava}
enum Boje {Crvena=1, Zelena=3, Plava=5}
```
- undefined / null vrednost - nevažeća vrednost (podtipovi svih tipova)
```
var novaGodina = null;
```
- any - promenljivoj možete dodeljivati bilo koji tip podataka
```
var nesto: any = "Tekstualni podatak";
var nesto = 909;
var nesto = true;
```
- void - kada nije potrebno dodeliti tip podatka (često kod funkcija)
```
function prazna(): void { document.write("Zdravo"); }
```



Tipovi podataka (3)

- Objekti - neprimitivni tipovi, to nisu stringovi, numerici, bulove vrednosti,...

```
declare function create(o: object | null): void;
create({ mojobjekat: 0 }); // OK
create(null); // OK
create(42); // Greska
create("moj tekst"); // Greska
create(false); // Greska
create(undefined); // Greska
```

- *Type assertions* - sličan operatoru cast u drugim jezicima
  "Kompajleru, veruj mi, znam šta radim"
  Pomoću zagrada:

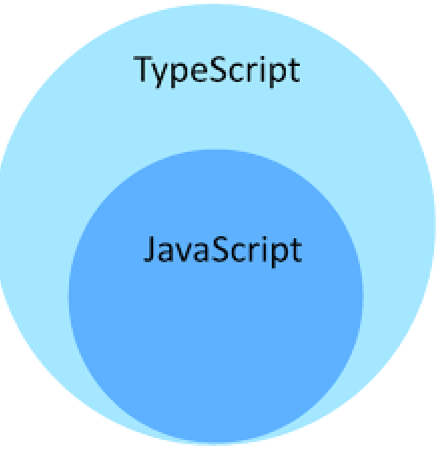
```
let tekst: any = "ETF BEOGRAD";
let duzina: number = (<string>tekst).length;
```

  Pomoću "*as*" sintakse:

```
let tekst: any = "ETF BEOGRAD";
let duzina: number = (tekst as string).length;
```

var, let, const

- *var* - deklaracija u programskom jeziku JavaScript
- *let* i *const* - dva nova tipa deklaracije promenljivih
- *let* sličan kao *var*, ali omogućava da se izbegnu neke klasične programerske greške
- *const* je proširenje *let*, jer sprečava ponovnu dodelu varijabli



Šta rešava let? (1)

```
function f(uslov: boolean) {
    if (uslov) {
        var x = 10;
    }
    return x;
}
f(true);
// vraca '10'

f(false);
// vraca 'undefined'
```

```
function f(uslov: boolean) {
    let a = 100;

    if (uslov) {
    // mogu da koristim a
        let b = a + 1;
        return b;
    }

    // Greska: b ovde ne postoji
    return b;
}
```

Opseg u blokovima

Šta rešava let? (2)

```
function f(x) {
    var x;
    var x;

    if (true) {
        var x;
    }
}
//ispravno
```

```
let x = 10;
let x = 20; // greska
```

```
function f(x) {
    let x = 100; // greska
}

function g() {
    let x = 100;
    var x = 100; // greska
}
```

Redeklarisanja / redefinisanja

Interfejsi

- Omogućavaju da postavimo strukturu za aplikaciju, ili strukture za objekte, funkcije, nizove, klase
- Mogu da se dodaju i opcione stavke (polja): atribut?: tipPodatka

Primer:

```
interface Osoba {
     ime: string;
    bojaKose: string;
    godine: number;
    zaposlen?: boolean;
}
```

Definisanje intefejsa za funkciju

Primer:

```
interface Dodavanje {
    (num1: number, num2: number)
}
var x: number = 5;
var y: number = 10;
var noviBroj: Dodavanje;
noviBroj = function(num1: number, num2: number){
    var rezultat: number = num1 + num2;
    document.write(rezultat);
    return rezultat;
}

var z = noviBroj(x,y);
```

Implementiranje klasa

Primer:

```
class Osoba{
    ime: string;
    godine: number;
    zaposlen: boolean = true;
    constructor(ime: string, godine?: number) {
            this.ime = ime;
            this.godine = godine;
    }
    otpusten(){
            this.zaposlen = false;
            return "Dao otkaz";
    }
}
var ja = new Osoba("Drazen",30);
```

Nasleđivanje klasa

Primer:

```
class Nastavnik extends Osoba {
    brojPredmeta: number = 3;

    dodajPredmet(){
          this.brojPredmeta++;
          return this.brojPredmeta;
    }
    oduzmiPredmet(){
          this.brojPredmeta--;
          return this.brojPredmeta;
    }
}
```

Moduli

- Moduli omogućavaju da se programski kod organizuje u više fajlova.
  Ovo skraćuje fajlove i čini ih održivim.
- Moduli se izvršavaju sa sopstvenim opsegom važenja, ne u globalnom opsegu (ovo znači da klase, varijable, funkcije, definisane u modulu, nisu vidljive van, ukoliko se ne izvezu).
- Odnosi između modula se postižu uvozom (*import*) i izvozom (*export*),
  na nivou fajla.
- Jedan modul uvozi neki drugi modul pomoću modula loader.
  Modul loader je u realnom vremenu odgovoran za lociranje i izvršavanje svih zavisnosti modula, pre nego što ga izvrši.

Export kod modula

- Svaka deklaracija (varijabla, funcija, klasa, interfejs) može biti izvezena (*export*).

<u>Primer:</u>

```
//Validation.ts
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
//ZipCodeValidator.ts
export const numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export * from "./ZipCodeValidator"; // eksport iz fajla
```

Import kod modula

- Slično kao i kod izvoza, radi se i uvoz fajla.

Primer:

```
import { ZipCodeValidator } from "./ZipCodeValidator";
let myValidator = new ZipCodeValidator();
```

Sa preimenovanjem:

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV()
```

Funkcije

- Funkcije u TS, slične kao funkcije u JS, ali postoje neke dodatne mogućnosti
- TS funkcije: dodati tipove parametrima koje prosleđujete, dodati tip rezultata funkciji (opciono)
- Primer:

```
function potpis(ime: string, prezime: string): string{
    return ime + ' ' + prezime;
}
```

- Opcioni parametri:
  nazivParametra?: tipPodatka
- Parametri funkcije sa podrazumevanim vrednostima:
  nazivParametra = "PodrazumevanaVred"

Angular (Angular 2+)

https://angular.io

Zašto Angular?

- Radni okvir za razvoj savremenih aplikacija za veb, mobilne i desktop platforme
- Čist i strukturiran pristup pisanju koda
- Dobra proširivost i ponovna upotreba
- Zasnovan na *HTML* i programskom jeziku *TypeScript*
- Veliki broj funkcija za upravljanje korisničkim unosom u pregledaču,
  za manipulisanje podataka na strani klijenta i za kontrolisanje kako su elementi prikazani u pregledaču
- Dobra kompatibilnost sa drugim tehnologijama (veza sa serverskom stranom ka Node.JS ili MVC .NET ...)
- Obezbeđena dobra struktura veb aplikacija kroz povezivanje podataka, injektiranje zavisnosti, HTTP komunikaciju

Projektni uzorci kod Angular

- Osim već pomenutih MVC i MVVM, Angular takođe podržava i promoviše korišćenje drugih projektivnih uzoraka koji su ključni za razvoj modernih veb aplikacija:
  - **Dependency Injection (DI)**: DI je projektivni uzorak koji je ključan za Angular, omogućavajući injektovanje zavisnosti u komponente, servise i druge klase. Ovo olakšava testiranje, održavanje i razmenu koda između komponenata.
  - **Singleton Pattern**: Singleton je projektivni uzorak koji garantuje da se određeni tip objekta instancira samo jednom tokom životnog ciklusa aplikacije. U Angular tehnologiji, servisi često koriste Singleton Pattern kako bi se osiguralo da se određena funkcionalnost ili podaci dele između različitih delova aplikacije.
  - **Facade Pattern**: Facade je projektivni uzorak koji pruža jednostavan interfejs za kompleksan sistem. U Angular-u, servisi se često koriste kao fasade za pristup kompleksnim ili spoljašnjim resursima, što olakšava korišćenje tih resursa u komponentama aplikacije.
  - **Observer Pattern**: Observer je projektivni uzorak koji omogućava objektima da se automatski obaveste kada se promeni stanje subjekta. U Angular-u, RxJS (*Reactive Extensions for JavaScript*) se često koristi za implementaciju Observer Pattern-a, omogućavajući reaktivno programiranje i upravljanje asinhronim događajima u veb aplikacijama.

Moduli

- Angular app koriste modularni dizajn - moduli nisu obavezni, ali se preporučuju!
- Ključne reči: **import** i **export**
- Za razliku od modula u prog. jeziku *TypeScript*, kod NgModules:
  - spoljni moduli se uvoze na vrhu fajla
  - funcije za izvoz se stavljaju na dnu fajla
- Sintaksa:
```
Import {Component} from 'angular2/core';
Export class MojaKlasa{ }
```
- NgModule može povezati komponente sa servisima
- Aplikacija uvek ima koreni modul (**AppModule**)
  koji obezbeđuje mehanizam pokretanja aplikacije
- Aplikacija tipično sadrži mnoštvo funkcionalnih modula

Direktive

- Direktive su JS klase sa metapodacima koje definišu strukturu i ponašanje. Obezbeđuju većinu UI funkcija za Angular aplikacije.
- Tri osnovna tipa direktiva su:
  - **Direktiva komponenata**:
    sadrži HTML šablon sa JS funkcijama za kreiranje samostalnog UI elementa,
    koji se može dodati u Angular aplikaciju;
    komponente su direktive koje se najčešće koriste.
  - **Strukturalne direktive**:
    direktive koje se koriste kada je potrebno da se manipuliše sa DOM-om;
    one omogućavaju da kreirate i uklonite element i komponentu iz prikaza.
  - **Atributske direktive**:
    one menjaju izgled i ponašanje HTML elemenata pomoću HTML atributa.

Komponente

- Svaka Angular aplikacija ima najmanje jednu (*root*) komponentu, koja povezuje hijerarhiju komponenti sa DOM (*document object model*) te veb stranice
- Svaka komponenta definiše klasu koja sadrži podatke aplikacije i logiku, i uvezana je sa HTML šablonom (*template*) tako da definiše pogled koji će biti prikazan
- Sinktaksa:

```
@Component({
    selector: 'app-hero-list',
    templateUrl:   './hero-list-component.html',
    providers:[ HeroService ]
})
```

Šabloni i povezivanje podataka

- Ugrađeno povezivanje podataka iz komponente sa elementima koji su prikazani na veb stranici (eng. *two-way data binding*)
- Direktive šablona obezbeđuju programsku logiku, a vezivanje povezuje podatke aplikacije i DOM
  - Kada se menjaju podaci na veb stranici, odnosno korisnik nešto unosi, ažuriraju se podaci u modelu (*event binding*)
  - Kada se menjaju podaci u modelu, automatski se ažuriraju sadržaj i novo izračunate vrednosti unutar veb stranice, odnosno podaci se interpoliraju u HTML (*property binding*)

Servisi i *Dependency injection*

- Za podatke ili logiku koja nije povezana sa određenim prikazom, a koju želimo da podelimo sa više različitih komponenti, kreiramo servisnu klasu (servis)
- *Dependency injection* – projektni uzorak/proces u kome jedna komponenta definiše zavisnost od drugih komponenata
- Definiciji servisne klase uvek prethodi dekorator @Injectable( )
- Dekorator pruža metapodatke koji omogućavaju servisu da ubaci u neku komponentu klijenta zavisnost
  (npr. komponenti koja pristupa veb serveru pomoću HTTP request, ubaciti HTTP servise u komponentu)

Rutiranje

- Ruter *NgModule* omogućava da definišemo putanju za navigaciju između različitih stanja aplikacije i hijerarhije pogleda u aplikaciji.
- Navigaciona konvencija:
  - Uneta URL adresa u veb pregledaču (adresnom polju) vodi do odgovarajuće veb stranice
  - Linkovi na veb stranicama vode do nove stranice
  - Korak nazad/korak napred navigira nas kroz istorijat veb stranica
- Kada ruter utvrdi da treutno stanje aplikacije zahteva određenu funkcionalnost, a modul koji ga definiše nije učitan, ruter može pokrenuti učitavanje modula na zahtev.
- Pravila navigacije: povezati navigacionu putanju sa vašim realizovanim komponentama

Dijagram Angular komunikacije

Module Component { }

Module Service { }

Module value 3.1415

Module Fn λ

Template

< >

Metadata

Directive { }

Property Binding

Metadata

Event Binding

Injector

Service

Component

Podela odgovornosti

- Pravila koja treba pratiti prilikom implementacije Angulara:
  - Prikaz fukcioniše kao zvanična prezentaciona struktura za aplikaciju.
    Sve logike aplikacije treba označiti kao direktive u HTML šablonu.
  - Ako želite da izvršite manipulaciju nad DOM, izvršite je samo u JavaScript kodu ugrađene ili prilagođene direktive
  - Implementirajte zadatke koje možete da izvršite više puta kao servise i dodajte ih u module pomoću uzorka *Dependency Injection*.
  - Uverite se da metapodaci odražavaju trenutno stanje modela i da predstavljaju jedinstveni izvor za podatke koje se koriste u prikazu.
  - Definišite kontrolere unutar modula imenskog prostora da biste lako mogli da izvršite "pakovanje aplikacije".

Kako Angular radi sa MVC? (1)

- **Model (M)**: predstavlja podatke i poslovnu logiku aplikacije
  - u Angularu predstavljeno servisima ili klasama koji upravljaju podacima, obavljaju operacije i komuniciraju sa API.
  - Model je odgovoran za preuzimanje, ažuriranje i manipulisanje podacima.
  - Primer servisa koji nam daje listu knjiga.

```
// book.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class BookService {
  private books = [
    { id: 1, title: 'Angular Basics', author: 'John Doe' },
    { id: 2, title: 'JavaScript Mastery', author: 'Jane Smith' },
    // ...more books
  ];

  getBooks() {
    return this.books;
  }
}
```

Kako Angular radi sa MVC? (2)

- **View (V)**: pogled predstavlja korisnički interfejs (UI) aplikacije
  - u Angularu pogled je predstavljen HTML šablonom koji definiše kako podaci iz Modela treba da se prikažu korisniku;
  - Pogled je odgovoran za prezentovanje podataka i dohvatanje korisničkog ulaza.

```
// book-list.component.ts
import { Component } from '@angular/core';
import { BookService } from './book.service';

@Component({
  selector: 'app-book-list',
  template: `
    <h1>Book List</h1>
    <ul>
      <li *ngFor="let book of books">{{ book.title }} by {{ book.author }}</li>
    </ul>
  `,
})
export class BookListComponent {
  books = [];

  constructor(private bookService: BookService) {
    this.books = bookService.getBooks();
  }
}
```

Kako Angular radi sa MVC? (3)

- **Controller (C)**: kontroler je u MVC uzorku veza između Modela i Pogleda. On prihvata ulazne korisničke podatke od Pogleda, procesira ih, interaguje sa Modelom da pročita podatke ili ih ažurira, i ažurira Pogled, u skladu sa tom akcijom.
  - U Angular tehologiji, uloga Kontrolera je u najvećom meri zamenjena komponentom.
  - U našem primeru to je klasa *BookListComponent.* Prihvataju se podaci iz modela (class *BookService*) i prikazuju na komponenti Pogled.

Kako Angular radi sa MVVM? (1)

- **Model (M)**: kao i kod MVC uzorka, Model je prikazan korišćenjem servisa ili klasa odgovornih za upravljanje podacima.

```
// counter.model.ts
export class CounterModel {
  value: number = 0;
}
```

Kako Angular radi sa MVVM? (2)

- **View (V)**:
  Angular šabloni se prikazuju kao Pogled, prikazuju podatke i dohvataju ulaz od strane korisnika. Povezivanje podataka (*binding*), interpolacija i direktive se koriste da uvežu *View* do *ViewModel* komponente.

```
// counter.component.ts
import { Component } from '@angular/core';
import { CounterModel } from './counter.model';

@Component({
  selector: 'app-counter',
  template: `
    <h1>Counter</h1>
    <p>Value: {{ counterModel.value }}</p>
    <button (click)="increment()">Increment</button>
    <button (click)="decrement()">Decrement</button>
  `,
})
export class CounterComponent {
  constructor(public counterModel: CounterModel) {}

  increment() {
    this.counterModel.value++;
  }

  decrement() {
    this.counterModel.value--;
  }
}
```

Kako Angular radi sa MVVM? (3)

- **ViewModel (VM)**: U Angularu ovaj element je prikazan komponentama i pridruženim TS klasama.
  - Klasa komponente sadrži aplikativnu logiku i interaguje sa servisima da isčita ili manipuliše podacima.
  - VM izlaže svojstva i metode, koje Pogled (View) povezuje. Ovo vezivanje se postiže mehanizmom povezivanja podataka (*data binding*), što omogućava nesmetani protok podataka između VM (komponente) i V (šabona).
  - U našem primeru:
    klasa *CounterComponent* interaguje sa modelom (klasa *CounterModel*) i otkriva vrednost brojača Pogledu.
    VM (*CounterComponent*) direktno se vezuje za model (*CounterModel*) da ažurira vrednost brojača i prikaže ga.

Angular CLI

- Dobar alat za proces izrade nove Angular aplikacije i instalaciju željenih biblioteka
- CLI brzo generiše nove komponente, direktive, procesne tokove, servise
- Jedan scenario:

```
npm install -g @angular/cli
ng new moja-aplikacija
cd moja-aplikacija
ng serve --open
```

- Otvara se http://localhost:4200
- https://angular.io/guide/quickstart

Najvažnije komande u Angular CLI

Komanda	Alt. naziv	Namena
ng new		Kreira novu Angular aplikaciju
ng serve		Gradi aplikaciju i pokreće je za testiranje
ng eject		Omogućava uređivanje fajlova webpack config
ng generate component [ime]	ng g c [ime]	Kreira novu komponentu
ng generate directive [ime]	ng g d [ime]	Kreira novu direktivu
ng generate module [ime]	ng g m [ime]	Kreia modul
ng generate pipe [ime]	ng g p [ime]	Kreira procesni tok
ng generate service [ime]	ng g s [ime]	Kreira servis
ng generate enum [ime]	ng g e [ime]	Kreira enumeraciju
ng generate guard [ime]	ng g g [ime]	Kreira zaštitu
ng generate interface [name]	ng g i [ime]	Kreira interfejs

Nova komponenta

- Komandom: ng generate component moja-komponenta
- Podrazumevano se kreira direktorijum željenog imena komponente i sledeći fajlovi:
  - Fajl komponente: **moja-komponenta.component.ts**
  - Fajl šablona: **moja-komponenta.component.html**
  - CSS fajl: **moja-komponenta.component.css**
  - Fajl za testiranje: **moja-komponenta.component.spec.ts**

Prikaz komponente

```
01 import {Component} from '@angular/core';
02 @Component ({
03   selector: 'moja-komponenta',
04   template: `
05       <h1>Zdravo!</h1>
06   `,
07  })
08 export class Chap3Component{
09   title = "Moja prva aplikacija";
10 }
```

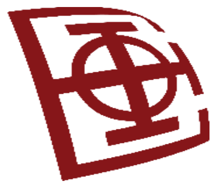
Definicija modula

- Modul označava kolekciju blokova programskog koda koji ima neku svrhu
- Angular koristi standardne JS module i takođe definiše Angular module – **NgModule**
- U JS ili ES, svaki fajl je modul i svi objekti definisani u fajlu, pripadaju tom modulu
- Objekti mogu biti izveženi (export), čime ih činimo javno vidljivim (public), a javno vidljive objekte možemo uvoziti (import) za upotrebu u drugim modulima
- Angular se isporučuje kao kolekcija JS modula
- Kolekcija JS modula se takođe referencira kao biblioteka
- Svaki naziv Angular biblioteke počinje prefiksom @angular
- Instalacija Angular biblioteka vrši se pomoću NPM alata za upravljanje paketima

Definicija dekoratora NgModule

- Dekorator @NgModule( ) piše se iznad definisane klase, i služi kao manifest za blok koda posvećen domenu aplikacije, radnom toku i blisko povezanom skupu mogućnosti.
- Kao i JS modul, NgModule može izvoziti funkcionalnosti koje bi bile korišćene od strane drugih NgModule-a.
- Metapodaci za NgModule klasu prikupljaju komponente, direktive i tokove, koje aplikacija koristi uporedo sa listom njih koje uvozi (import) ili izvozi (export)

- Svaka Angular aplikacija ima koreni NgModule, koji se po konvenciji naziva **AppModule** i nalazi se u fajlu *app.module.ts*



Upotreba dekoratora NgModule

```
//fajl: app.module.ts

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { MojaKomponentaComponent } from './moja-komponenta/moja-komponenta.component';

@NgModule({
      declarations: [    AppComponent,    MojaKomponentaComponent  ],
      imports: [    BrowserModule,    AppRoutingModule  ],
      providers: [],
      bootstrap: [AppComponent]
})
export class AppModule { }
```

Razumevanje ključnih svojstava dekoratora NgModule (1)

- `declarations` – svojstvo koje koristimo za deklarisanje niza komponenti, direktiva i/ili procesnih tokova, koji pripadaju trenutnom (lokalnom) modulu. Sve što je ovde deklarisano dostupno je unutar samog modula i mogu se koristiti u svim njegovim šablonima (*templates*).
- `imports` – svojstvo koje se koristi za uvoz elemenata drugih modula u okviru trenutnog modula. Uvozom drugih modula i njegovih elemenata (komponenti, direktiva, servisa,...) koji su definisani u njemu, oni postaju vidljivi u trenutnom modulu.
- `exports` – svojstvo koje se koristi za izvoz komponenti, direktiva, ili procesnih tokova, koje želimo da delimo sa drugim modulima. Neophodno je da ti drugi moduli urade uvoz (svojstvo *import*) ovog trenutnog modula.
- `providers` – svojstvo koje se koristi za registrovanje provajdera servisa unutar modula. Servisi koji su registrovani ovde su dostupni za injektovanje u komponente, direktive i druge servise unutar ovog modula i njegovih podmodula.
- `bootstrap` – svojstvo koje se koristi samo u korenom modulu aplikacije i označava glavnu komponentu koja će biti pokrenuta prilikom inicijalizacije aplikacije (Komponenta koja se prvo renderuje nakon pokretanja aplikacije).

Razumevanje ključnih svojstava dekoratora NgModule (2)

- `entryComponents` – svojstvo koje se koristi za definisanje komponenti koje se dinamički instanciraju, obično preko Angular reflektora komponenti. Komponente koje su navedene u ovom svojstvu su van stabla komponenti, što znači da neće biti direktno referencirane u šablonima drugih komponenti, već će biti dinamički kreirane kroz programsku logiku. <u>Primer korišćenja</u>: kada su komponente kreirane dinamički na osnovu korisničke interakcije ili na osnovu podataka sa servera.
- `schemas` – svojstvo koje se koristi za definisanje šema koje se primenjuju na komponente u modulu. Šeme definišu HTML kod koji se može koristiti u šablonima komponenti. <u>Primer korišćenja</u>: Ako želite da ograničite korišćenje određenih HTML elemenata ili atributa, možete ih onda navesti u šemi. Ovo pomaće u zaštiti aplikacija od XSS (*Cross-Site Scripting*) napada i pruža dodatni sloj sigurnosti!
- `id` – svojstvo koje se koristi za dodeljivanje jedinstveno ID modulu. Iako se ne koristi često, ovo može biti korisno u scenarijima gde je potrebno jedinstveno identifikovati modul, npr. u dijagnostičkim ili analitičkim alatima, koji analiziraju strukturu Angular aplikacije.

Kreiranje pokretača Angulara

```
//fajl: main.ts

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
   enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
   .catch(err => console.error(err));
```

Optimizacija proizvodne aplikacije

Pokretanje aplikacije pomoću modula AppModule

Angular

Angular komponente

Šta su komponente?

- Angular komponente - gradivni blokovi koji se koriste za kreiranje aplikacije
- Omogućavaju da se naprave samostalni UI elementi i da se kontroliše izgled i funkcije pomoću *TypeScript* koda i *HTML* šablona
- Angular komponenta sadrži 2 dela: dekorator (@Component) i klasu (class)
- Dekorator služi za konfigurisanje komponente,
  uključujući naziv selektora, HTML šablon (obično samo vezu do njega – *templateUrl*) i opciono CSS šablon (ili samo vezu do njega – *stylesUrls*)
- Klasa omogućava da komponenti dodelite logiku, ponašanje i rukovaoce događaja i da je izvezemo u druge *TypeScript* fajlove (export opcija)

Primer i konfiguracija komponente

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

Prikaz opcija kod komponente

- **selector –** definiše naziv HTML taga koji se koristi za dodavanje komponente u aplikaciju pomoću HTML
- **template** ili **templateUrl**
  - **template** – dodavanje ugrađene HTML da bismo definisali izgled komponente, korisna kada nemamo mnogo koda i dodatnih fajlova
  - **templateUrl** – služi za uvoz spoljnih šablona, umesto ugrađenog HTML (korisno za izdvajanje velike količine HTML koda)
- **styles** ili **stylesUrls**
  - **styles** – dodavanje ugrađenog CSS u komponentu
  - **stylesUrls** – omogućava uvoz niza spoljnih CSS opisa stilova
- **viewProviders –** niz provajdera za injektiranje zavisnosti, koji služi da koristimo već razvijene Angular servise

Definisanje selektora

- Primena komponente u HTML stranicama
- Naziv selektora se upotrebljava kao naziv HTML taga
- Nije dozvoljeno da koristimo razmake u nazivu selektora
- Primer:

```
@Component({
   selector: 'moja-komponenta'
})
```

Primena u HTML:

```
<moja-komponenta> </moja-komponenta>
```

Izrada šablona

- Služe da definišu izgled Angular komponente, pišu se kao HTML
- Angular omogućava da se koriste ugrađeni šabloni ili spoljne datoteke šablona
- Šablon se dodaje u dekorator @Component
- Jednoredni šablon može da koristi jednostruke (apostrofe) ili dvostruke navodnike, a višeredni šablon koristi obrnute navodnike ` `
- Na ugrađene CSS stilove se primenjuju ista pravila kao i na standardne CSS šablone

```
styles: [`
   p {
      color: yellow;
      font-size: 25px;
   }
`]
```

Upotreba konstruktora

- Konstruktori – za dodelu podrazumevanih vrednosti komponente
- Kada se promenljive koriste u komponenti, uvek će biti inicijalizovane
- Primer sintakse:

```
export class Student {
    name: string;
    constructor(){
        this.name = "Drazen";
    }
}
```

Primer komponente sa konstruktorom

```
import {Component} from '@angular/core';
@Component ({
   selector: 'simple-constructor',
   template: `
         <p>Danas je {{today}}!</p>
   `

})
export class KoriscenjeKonstruktora{
    today: Date;
    constructor(){
         this.today = new Date();
    }
}
```

Upotreba spoljnih šablona

- Upotreba zasebnih datoteka u okviru naše komponente i njenog dekoratera
- Prednost: lakše se određuje namena datoteke, lakše čitanje komponente, manji broj linija koda
- Primer:

```
@Component({
      selector: 'login-komponenta',
      templateUrl: "login-komponenta.component.html",
      styleUrl: ["./stilovi1.css",
                 "./stilovi2.css"
      ]
})
```

Injektiranje direktiva

- Injektiranje zavisnosti definiše i dinamički injektira objekat zavisnosti u drugi objekat, pa sve funkcije koje obezbeđuje objekat zavisnosti postaju dostupne.
- Angular objezbeđuje injektiranje zavisnosti pomoću provajdera i servisa injektora.
- Da bi se koristilo injektiranje zavisnosti na drugoj direktivi ili komponenti, potrebno je da se u okviru modula za aplikaciju doda naziv klase direktive ili komponente u metapodetke *declarations* u dekoratoru @*NgModule*, čime se niz direktiva uvozi u aplikaciju

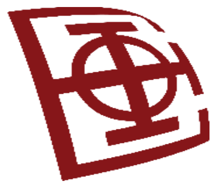
Prosleđivanje podataka pomoću injektiranja zavisnosti

- Da bi se u Angularu uneli podaci u drugu direktivu ili komponentu, potrebno je da se uveze dekorator Input iz paketa @angular/core:

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

- Kada se uveze ovaj decorator, možemo da počnemo da definišemo podatke koje ćemo da unesemo u direktivu.
  Definisanje dekoratora se radi korišćenjem @input(),
  koji koristi string podatak kao parametar.
- Primer:

```
@Input('name') personName: string;
```



Primer

```
import { Component } from '@angular/core';
import {myInput} from './input.component';
@Component({
   selector: 'app-root',
   template: `
   <myInput name="Brendan" occupation="Student/Author"></myInput>
   <myInput name="Brad" occupation="Analyst/Author"></myInput>
   <myInput name="Caleb" occupation="Student/Author"></myInput>
   <myInput></myInput>
   `

})
   export class AppComponent {
   title = 'Using Inputs in Angular';
}
```

Primer

```
01 import {Component, Input}
          from '@angular/core';
02 @Component ({
03 selector: "myInput",
04 template: `
05 <div>
06    Name: {{personName}}
07    <br/>
08     Job: {{occupation}}
09 </div>
10 `,
11 styles: [`
12 div {
13    margin: 10px;
14    padding: 15px;
15    border: 3px solid grey;
16 }
17 `]
18 })
19 export class myInput {
20 @Input('name') personName: string;
21 @Input('occupation’) occupation: string;
22 constructor() {
23  this.personName = "John Doe";
24  this.occupation = "Anonymity"
25  }
26 }
```

Životni ciklus komponente (ili direktive)

- Za svaku komponentu/direktivu postoji veći broj interfejsa kojim možemo tu komponentu kreirati, ažurirati, uništiti. Svaki interfejs ima svoju metodu sa udicom (hook) koja ima prefiks **ng**.
- Angular u sledećem redosledu pokreće svoje metode sa udicom:

R.br.	Naziv metode	Opis
1.	ngOnChanges	Metoda kada se ulazne ili izlazne vrednosti menjaju (*data binding*)
2.	ngOnInit	Metoda koja se pokreće nakon prvog *ngOnChanges*
3.	ngDoCheck	Detekcija prilagođenih promena od strane programera
4.	ngAfterContentInit	Nakon što se sadržaj komponente inicijalizuje
5.	ngAfterContentChecked	Nakon svake provere sadržaja komponente
6.	ngAfterViewInit	Nakon što se pogled komponente inicijalizuje
7.	ngAfterViewChecked	Nakon svake provere pogleda komponente
8.	ngOnDestroy	Neposredno pre nego što je direktiva uništetna

Angular

Angular izrazi i filteri

Upotreba izraza (1)

- Najjednostavniji način za prikaz podataka iz komponente u prozor Angulara je korišćenje izraza. Izrazi su blokovi koda, koji su smešteni unutar dvostrukih vitičastih zagrada:
  {{expression}}
- Angular kompajler kompajlira izraz kao HTML elemente, pa će korisniku biti prikazani rezultati izraza.
- Primeri:
  {{1+5}}
  {{'ETF' + 'Beograd'}}
- Rezultati izvršavanja ovih izraza:
  6
  ETFBeograd

Upotreba izraza (2)

- Izrazi su povezani sa modelom podataka
- Prva pogodnost:
  možemo koristiti nazive i funkcije koje su definisane u komponenti unutar izraza.
- Druga pogodnost:
  izrazi povezani sa komponentom, pa se menjaju kada se menjaju i podaci u komponenti.
- Primer:
  name: string='Drazen';
  score: number=95;
- Direktno ukazivanje na vrednosti *name* i *score*:
  Ime: {{name}}
  Ukupno: {{score}}
  Ukupno sa bonusom: {{score+5}}

Sličnost sa TypeScript/JavaScript

- Angular izrazi se razlikuju od TS/JS izraza u sledećem:
  - Izračunavanje atributa - nazivi svojstva u Angularu se izračunavaju prema modelu komponente, umesto prema imenskom prostoru JavaScript.
  - Angular izrazi su jednostavniji za upotrebu – oni ne generišu greške kada se susretnu sa nedefinisanim ili nevažećim tipom promenljivih, već ih tretiraju kao promenljive koje nemaju vrednost.
  - Angular izrazi ne kontrolišu protok - izrazi onemogućavaju sledeće:
    - dodele (npr. =, +=, -=)
    - operator new
    - uslove
    - petlje
    - inkrementiranje i dekrementiranje operatora (++ i --)
  - Ne mogu se generisati greške unutar izraza.

Izrazi kao stringovi

- Angular izračunava izraze kao stringove koji se koriste za definisanje vrednosti direktiva.
- Kada bi se postavila vrednost direktive *ng-click* u šablon, ona određuje izraz
- Unutar izraza može da se ukaže na promenljivu komponente ili upotrebi neka druga sintaksa izraza, na primer:

```
<span ng-click="myFunction()"></span>
<span ng-click="myFunction(var, 'stringParameter')"></span>
<span ng-click="myFunction(5*var)"></span>
```

- Kako izrazi šablona mogu da pristupe komponenti, znači da je i moguća izmena komponente unutar Angular izraza
- Primer direktive (click) koja menja vrednost msg unutar modela komponente:

```
<span (click)="msg='clicked'"></span>
```

Primer upotrebe izraza

```
import { Component } from '@angular/core';
@Component({
        selector: 'app-root',
        template: `
          <h1>Expressions</h1>
          Number:<br>
          {{5}}<hr>
          String:<br>
          {{'My String'}}<hr>
          Adding two strings together:<br>
          {{'String1' + ' ' + 'String2'}}<hr>
          Adding two numbers together:<br>
          {{5+5}}<hr>
          Adding strings and numbers together:<br>
          {{5 + '+' + 5 + '='}}{{5+5}}<hr>
          Comparing two numbers with each other:<br>
          {{5===5}}<hr>
        `,
})
export class AppComponent {}
```

Expressions

Number:
5

String:
My String

Adding two strings together:
String1 String2

Adding two numbers together:
10

Adding strings and numbers together:
5+5=10

Comparing two numbers with each other:
true

Interakcija izraza u @Component sa klasom

```
import { Component } from '@angular/core';
@Component({
		selector: 'app-root',
		template: `
				Direktno procitane promenvljive u komponenti: <br>
				{{brzina}} {{vozilo}}<hr>
				Dodavanje promenljive u komponentu:<br>
				{{brzina + ' ' + vozilo}}<hr>
				Pozivanje funkcije u komponenti:<br>
				{{smanji(brzina)}} {{povecaj('Dzip')}}<hr>
				<a (click)=setValues('Fast', novoVozilo)">
				  Pritisni da promeniš na Brzo {{novoVozilo}}</a><hr>
				<a (click)="setNoveVrednosti(novaBrzina, 'Avion')">
				  Pritisni da promeniš brzinu vozilu {{novaBrzina}} Yugo</a><hr>
				<a (click)="vozilo='Auto'">
				  Pritisni da promeniš vrstu vozila na Auto </a><hr>
				<a (click)="vozilo='Unapredjen ' + vozilo">
				  Pritisni da unaprediš vozilo!</a><hr>
		`,
		styles: [`
				a{color: blue; text-decoration: underline; cursor: pointer}
		`]
})
```

Interakcija izraza u @Component sa klasom

```
export class AppComponent {
   brzina = 'Slow';
   vozilo = 'Voz';
   novaBrzina = 'Najbrza';
   novoVozilo = 'Tramvaj';
   povecaj = function(str: any) {
               str = str.toUpperCase();
               return str;
   }
   smanji = function(str: any) {
               return str.toLowerCase();
   }
   setNoveVrednosti = function(brzina: any, vozilo: any){
               this.brzina = brzina;
               this.vozilo = vozilo;
   }
}
```

Upotreba TypeScript u Angular izrazima

```
import { Component } from '@angular/core';
@Component({
        selector: 'app-root',
        template: `
                <h1>Expressions</h1>
                Array: <br>{{myArr.join(', ')}}<br/><hr>
                Elements removed from array:<br/>    {{removedArr.join(', ')}}<hr>
                <a (click)="myArr.push(myMath.floor(myMath.random()*100+1))">
                        Pritisni da dodaš vrednost u niz! </a><hr>
                <a (click)="removedArr.push(myArr.shift())"> Pritisni da skloniš prvi element iz niza</a><hr>
                Veličina niza:<br> {{myArr.length}}<hr>
                Maksimalan broj uklonjen iz niza:<br> {{myMath.max.apply(myMath, removedArr)}}<hr>
                `,
        styles: [`
                a { color: blue; cursor: pointer; }
              `],
})
export class AppComponent {
        myMath = Math;
        myArr: number[] = [1];
        removedArr: number[] = [0];
}
```

TypescriptExpressions

localhost:4200

Expressions

Array:
1

Elements removed from array:
0

Click to append a value to the array

Click to remove the first value from the array

Size of Array:
1

Max number removed from the array:
0

TypescriptExpressions

localhost:4200

Expressions

Array:
1, 29, 44, 3, 100

Elements removed from array:
0

Click to append a value to the array

Click to remove the first value from the array

Size of Array:
5

Max number removed from the array:
0

TypescriptExpressions

localhost:4200

Expressions

Array:
44, 3, 100

Elements removed from array:
0, 1, 29

Click to append a value to the array

Click to remove the first value from the array

Size of Array:
3

Max number removed from the array:
29

Filteri (eng. *pipe*)

- Filteri (od verzije Angular 2 zvanično „Cevi“ – eng. *Pipes*) predstavljaju način da transformišemo string podatak, novčane iznose, datume ili druge tipove podataka koje želimo da prikažemo u veb pregledaču.
- Filteri su jednostavne funkcije koje se koriste u izrazima šablona za prihvatanje ulazne vrednosti, i vraćanje transformisane vrednosti.
- Filteri su korisni jer se mogu koristiti u celoj aplikaciji, dok svaki filter deklariše samo jednom.
- Na primer datum *March 28, 2024*, želimo da prikažemo baš kao takav oblik datuma ili kao kratak datum (3/28/2024), umesto neobrađenog string formata.
- Primer: **{{ startDate | date: 'shortDate'}}** , a za početni string stavimo neobrađeni: “Thu Mar 28 2024 14:59:33 GMT+0200 (Central European Daylight Time)”, prikazuje formatiran ispis: 3/28/2024

Upotreba vertikalnih crta

- Vertikalne crte se implementiraju unutar izraza interpolacije, pomoću sledeće sintakse: {{ *expression* | *pipe_name* }}
- Na ovaj način izražavamo želju da primenimo određeni filter *pipe_name* nad našim izrazom *expression*.
- Ako spojimo više operatora vertikalne crte i navedemo više filtera, oni će se izvršavati po redosledu po kojem smo ih odredili:
  {{ *expression* | *pipe1* | *pipe2* }}
- Neki filteri omogućavaju da unosi budu u obliku parametara funkcije:
  {{ *expression* | *pipe:parameter1:parameter2* }}
- Angular omogućava nekoliko tipova operatora vertikalne crte koji omogućavaju da se lako formatiraju stringovi, objekti i nizovi u šablonima komponente.

Ugrađeni filteri (eng. *Built-in pipes*)

- **DatePipe**: Filter koji formatira vrednost datuma prema lokalnim pravilima.
- **UpperCasePipe**: Filter koji transformiše tekst u sva velika slova.
- **LowerCasePipe**: Filter koji transformiše tekst u sva mala slova.
- **TitleCasePipe**: Filter koji transformiše prva slova svake reči u velika, a sva ostala u mala slova.
- **CurrencyPipe**: Filter koji transformiše brojeve u valute.
- **DecimalPipe**: Filter koji transformiše brojeve u string-ove sa decimalnom tačkom, formatirane prema lokalnim pravilima.
- **PercentPipe**: Filter koji transformiše brojeve u procentualni string.
- **AsyncPipe**: Pretplaćuje se ili otkazuje pretplatu na asinhroni izvor, kao što to radi opservabla.
- **JsonPipe**: Filter koji konvertuje bilo koji tip (*any*) u format *JSON* string-a.

Ugrađeni filteri (*Built-in pipes*)

Filter	Description
`currency[ :currencyCode?[ :symbolDisplay?[ :digits?]]]`	Formats a number as currency, based on the `currencyCode` value provided. If no `currencyCode` value is provided, the default code for the locale is used. Here is an example: `{{123.46 | currency:"USD" }}`
`json`	Formats a TypeScript object into a JSON string. Here is an example: `{{ {'name':'Brad'} | json }}`
`slice:start:end`	Limits the data represented in the expression by the indexed amount. If the expression is a string, it is limited in the number of characters. If the result of the expression is an array, it is limited in the number of elements. Consider these examples: `{{ "Fuzzy Wuzzy" | slice:1:9 }}` `{{ ['a','b','c','d'] | slice:0:2 }}`
`lowercase`	Outputs the result of the expression as lowercase.
`uppercase`	Outputs the result of the expression as uppercase.
`number[:pre.post-postEnd]`	Formats the number as text. If a `pre` parameter is specified, the number of whole numbers is limited to that size. If `post-postEnd` is specified, the number of decimal places displayed is limited to that range or size. Consider these examples: `{{ 123.4567 | number:1.2-3 }}` `{{ 123.4567 | number:1.3 }}`

Ugrađeni filteri (*Built-in pipes*)

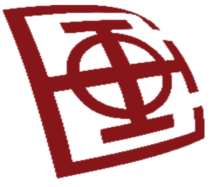
`date[:format]`	Formats a TypeScript date object, a timestamp, or an ISO 8601 date string, using the *format* parameter. Here is an example: `{{1389323623006 | date:'yyyy-MM-dd HH:mm:ss Z'}}` The *format* parameter uses the following date formatting characters: ▪ **yyyy**: Four-digit year ▪ **yy**: Two-digit year ▪ **MMMM**: Month in year, `January` through `December` ▪ **MMM**: Month in year, `Jan` through `Dec` ▪ **MM**: Month in year, padded, `01` through `12` ▪ **M**: Month in year, `1` through `12` ▪ **dd**: Day in month, padded, `01` through `31` ▪ **d**: Day in month, `1` through `31` ▪ **EEEE**: Day in week, `Sunday` through `Saturday` ▪ **EEE**: Day in Week, `Sun` through `Sat` ▪ **HH**: Hour in day, padded, `00` through `23` ▪ **H**: Hour in day, `0` through `23` ▪ **hh** or `jj`: Hour in a.m./p.m., padded, `01` through `12` ▪ **h** or `j`: Hour in a.m./p.m., `1` through `12` ▪ **mm**: Minute in hour, padded, `00` through `59` ▪ **m**: Minute in hour, `0` through `59` ▪ **ss**: Second in minute, padded, `00` through `59` ▪ **s**: Second in minute, `0` through `59` ▪ **.sss** or **,sss**: Millisecond in second, padded, `000-999` ▪ **a**: a.m./p.m. marker ▪ **Z**: Four-digit time zone offset, `-1200` through `+1200` The *format* string for `date` can also be one of the following predefined names: ▪ **medium**: Same as `'yMMMdHms'` ▪ **short**: same as `'yMdhm'` ▪ **fullDate**: same as `'yMMMMEEEEd'` ▪ **longDate**: same as `'yMMMMd'` ▪ **mediumDate**: same as `'yMMMd'` ▪ **shortDate**: same as `'yMd'` ▪ **mediumTime**: same as `'hms'` ▪ **shortTime**: same as `'hm'` The format shown here is `en_US`, but the format always matches the locale of the Angular application.
`async`	Waits for a promise and returns the most recent value received. It then updates the view.

Ugrađeni filteri: Primer filtera za datum (*DatePipe*)

```
@Component({
        selector: 'date-pipe',
        template: `<div>
                <p>Today is {{today | date}}</p>
                <p>Or if you prefer, {{today | date:'fullDate'}}</p>
                <p>The time is {{today | date:'h:mm a z'}}</p>
            </div>`
        })

        export class DatePipeComponent {
                today: number = Date.now();
        }
```

Ugrađeni filteri: Primer filtera malih i velikih slova

```
@Component({
        selector: 'lowerupper-pipe',
        template: `<div> <label>Name: </label>
                        <input #name (keyup)="change(name.value)" type="text" />
                        <p>In lowercase:</p>
                        <pre>'{{ value | lowercase }}'</pre>
                        <p>In uppercase:</p>
                        <pre>'{{ value | uppercase }}'</pre>
                  </div>`,
        })
        export class LowerUpperPipeComponent {
                value: string = '';
                change(value: string) {
                        this.value = value;
                }
        }
```



Ugrađeni filteri: Primer filtera *TitleCasePipe*

```
@Component({
  selector: 'titlecase-pipe',
  template: `<div>
    <p>{{ 'drazen draskovic' | titlecase }}</p>
    <!- ocekivani izlaz: "Drazen Draskovic" -->
    <p>{{ 'tHIs is mIXeD CaSe' | titlecase }}</p>
    <!-- ocekivani izlaz: "This Is Mixed Case" -->
    <p>{{ "it's non-trivial question" | titlecase }}</p>
    <!-- ocekivani izlaz: "It's Non-trivial Question" -->
    <p>{{ 'one,two,three' | titlecase }}</p>
    <!-- ocekivani izlaz: "One,two,three" -->
    <p>{{ 'true|false' | titlecase }}</p>
    <!-- ocekivani izlaz: "True|false" -->
    <p>{{ 'foo-vs-bar' | titlecase }}</p>
    <!-- ocekivani izlaz: "Foo-vs-bar" -->
  </div>`,
})
export class TitleCasePipeComponent {}
```

Ugrađeni filteri: Primer JSON filtera (*JsonPipe*)

```
@Component({
  selector: 'json-pipe',
  template: `<div>
    <p>Bez filtera za JSON:</p>
    <pre>{{ object }}</pre>
    <p>Sa filterom za JSON:</p>
    <pre>{{ object | json }}</pre>
  </div>`,
})
export class JsonPipeComponent {
  object: Object = {foo: 'bar', baz: 'qux', nested: {xyz: 3, numbers: [1,2,3,4,5]}};
}
```

Angular

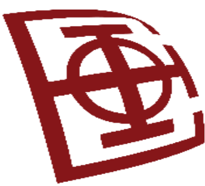
Povezivanje podataka

Povezivanje podataka

- Proces povezivanja podataka iz komponente, sa onim što se prikazuje na veb stranici
- Kada se podaci u komponenti izmene => korisnički interfejs koji se prikazuje korisniku se automatski ažurira
- Model je jedini uvek izvor podataka, prikaz je samo projekcija modela

Tipovi povezivanja

- interpolacija (*interpolation*) - koristimo duple vitičaste zagrade {{ }}
  za dobijanje vrednosti direktno iz klase *Component*
- povezivanje svojstava (*property binding*) - koristimo za postavljanje HTML elemenata
- povezivanje događaja (*event binding*) - koristimo za upravljanje korisničkim unosima
- povezivanje atributa (*attribute binding*) - omogućava da podesimo atribute HTML elemenata
- povezivanje klasa (*class binding*) - koristimo za postavljanje naziva CSS klase elementa
- povezivanje stilova (*style binding*) - koristimo za kreiranje ugrađenih CSS stilova elementa
- dvosmerno povezivanje (*two way binding*) pomoću direktive ngModel - koristimo u obrascima za unos podataka radi prijema i prikaza podataka



Interpolacija

- Koristi se {{ }} za izračunavanje izraza šablona
- Interpolacija može biti ugrađena u izvorni kod ili može da bude referenca svojstva klase
- Primer sintakse za zadavanje vrednosti:
  `<img src="{{slikaURL}}"/>`
- Primer interpolacije u kojoj se koriste stringovi i funkcija

```
import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   template: `
   {{str1 + ' ' + name}}
   <br>
   <img src="{{imageSrc}}" />
   <br>
   <p>{{str2 + getLikes(likes)}}</p>
   `,
   styles: [`
   img{
    width: 300px;
    height: auto;
   }
   p{
    font-size: 35px;
    color: darkBlue;
   }
   `]
 })
```

```
export class AppComponent {
   str1: string = "Hello my name is"
   name: string = "Drazen"
   str2: string = "I like to"
   likes: string[] = ['hike', 'swim',
                      'play piano']
   getLikes = function(arr: any){
     var arrString = arr.join(", ");
     return " " + arrString
   }
   imageSrc: string =
    "../assets/images/mojaSlika.jpg"
}
```

Povezivanje svojstava

- Povezivanje svojstava se koristi kada treba da postavite svojstvo HTML elementa.
- Svojstvo ćete postaviti tako što se definiše vrednost koju želite u okviru klase *Component*.
- Često interpolacija može da postigne isti rezultat kao i prilikom povezivanja svojstava.
- Primer povezivanja svojstava i primene naziva klase

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `
   <img [src]="myPic"/>
   <br>
   <button [disabled]="isEnabled">Click me</button><hr>
   <button disabled="{!isEnabled}">Click me</button><br>
   <p [ngClass]="className">This is cool stuff</p>
`,
styles: [`
   img {
     height: 100px;
     width auto;
   }
   .myClass {
     color: red;
     font-size: 24px;
   }
`]
})
```

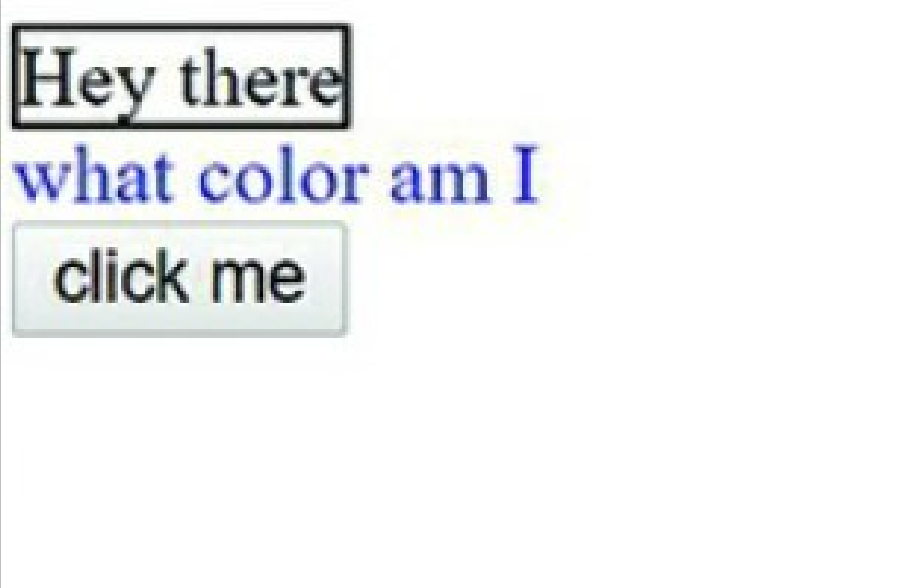
```
export class AppComponent {
   myPic: string = "../assets/images/sunset.JPG";
   isEnabled: boolean = false;
   className: string = "myClass";
}
```

Povezivanje stilova

- Definiše se CSS svojstvo stila u zagradama sa izrazom dodele koji se stavlja pod navodnike.
- Sintaksa je slična za povezivanje klase, ali se koristi prefiks *style*, umesto prefiksa *class.*
- *Primer:*

```
<p [style.styleProperty] = "assignment"></p>

<div [style.backgroundColor] = "'green'"></div>
```

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `
    <span [style.border]="myBorder">Hey there</span>
    <div [style.color]="twoColors ? 'blue' : 'forestgreen'">
      what color am I
    </div>
    <button (click)="changeColor()">click me</button>
`
})
export class AppComponent {
   twoColors: boolean = true;
   changeColor = function(){
     this.twoColors = !this.twoColors;
   }
   myBorder = "1px solid black";
}
```



Povezivanje događaja

- Koristi se za upravljanje korisničkim unosima, kao što su: pritisak (“click”) na dugme, pritisci na taster tastature i pomeranje miša.
- Povezivanje događaja je kao povezivanje HTML-a i JavaScript-a samo što se uklanja prefiks "*on*" iz povezivanja i što se događaj postavlja u zagradama, npr.: (click) ili (keyup).
- Povezivanje događaja služi za pokretanje funkcija iz komponente.
- Primer:

```
<button (click)="mojaFunkcija()">Dugme</button>
```

Primer (1. deo)

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

@Component({
selector: 'app-root',
template: `
   <div (mousemove)="move($event)">
   <img [src]="imageUrl"
   (mouseenter)="mouseGoesIn()"
   (mouseleave)="mouseLeft()"
   (dblclick)="changeImg()" /><br>
   double click the picture to change it<br>
   The Mouse has {{mouse}}<hr>
   <button (click)="changeImg()">Change Picture</button><hr>
   <input (keyup)="onKeyup($event)"
   (keydown)="onKeydown($event)"
   (keypress)="keypress($event)"
   (blur)="underTheScope($event)"
   (focus)="underTheScope($event)">
   {{view}}
```

Primer (2. deo)

```
 <p>On key up: {{upValues}}</p>
   <p>on key down: {{downValues}}</p>
   <p>on key press: {{keypressValue}}</p>
   <p (mousemove)="move($event)">
       x coordinates: {{x}}
   <br> y coordinates: {{y}}
   </p>
   </div>
`,
styles: [`
   img {
     width: auto;
     height: 300px;
   }
`]
})
```

```
export class AppComponent {
   counter = 0;
   mouse: string;
   upValues: string = '';
   downValues: string = '';
   keypressValue: string = "";
   x: string = "";
   y: string = '';
   view: string = '';
mouseGoesIn = function(){
   this.mouse = "entered";
};
mouseLeft = function(){
   this.mouse = "left";
}
imageArray: string[] = [
   "../assets/images/flower.jpg",
   "../assets/images/lake.jpg",
   //extensions are case sensitive
   "../assets/images/bison.jpg",
]
```

Primer (3. deo)

```
imageUrl: string = this.imageArray[this.counter];
   changeImg = function(){
     if(this.counter < this.imageArray.length - 1){
        this.counter++;
     }else{
        this.counter = 0;
     }
     this.imageUrl=this.imageArray[this.counter];
   }
onKeyup(event:any){
   this.upValues = event.key;
   //this.upValues += event.target.value + ' | ';
}
onKeydown(event:any){
   this.downValues = event.key;
   //this.downValues += event.target.value + " | ";
}
keypress(event:any){
   this.keypressValue = event.key;
   //this.keypressValue += event.target.value + " | ";
}
```

Primer (4. deo)

```
move(event:any){
   this.x = event.clientX;
   this.y = event.clientY;
}
underTheScope(event:any){
   if(event.type == "focus"){
     this.view = "the text box is focused";
   }
   else if(event.type == "blur"){
     this.view = "the input box is blurred";
   }
   console.log(event);
   }
}
```

Event

localhost:4200

double click the picture to change it
The Mouse has entered

Change Picture

Event

localhost:4200

double click the picture to change it
The Mouse has entered

Change Picture

Event

localhost:4200

double click the picture to change it
The Mouse has entered

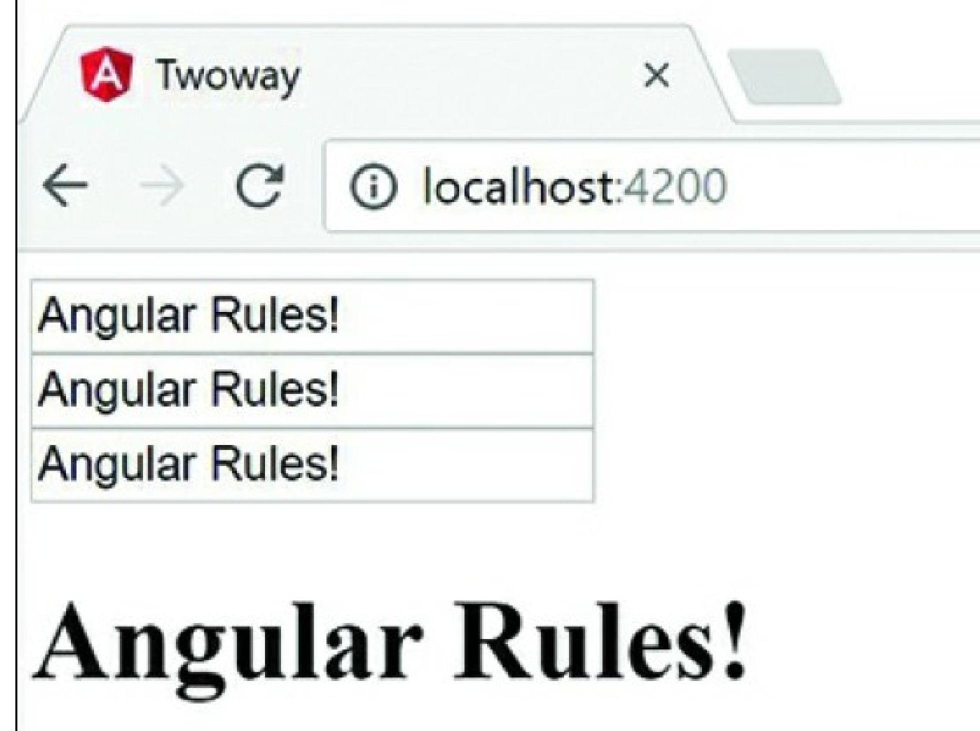
Change Picture

Dvosmerno povezivanje

- Dvosmerno povezivanje - istovremeno se podaci prikazuju i ažuriraju
- Podaci se prikazuju pomoću direktive *ngModel*
- Sintaksa za dvosmerno povezivanje:
  `<input [(ngModel)] = "vrednost">`

Primer

```
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
template: `
    <input [(ngModel)]="text"><br>
    <input bindon-ngModel="text"><br>
    <input [value]="text"
(input)="text=$event.target.value">
    <h1>{{text}}</h1>
`
})
export class AppComponent {
    text: string = "Angular Rules!";
}
```

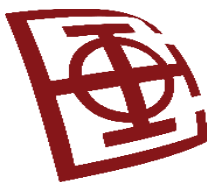


Angular

Ugrađene i prilagođene direktive

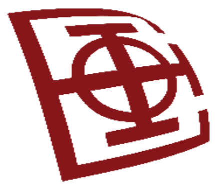
Razumevanje direktiva

- Direktive predstavljaju kombinaciju *Angular* oznaka šablona i *TypeScript* koda
  - Angular direktive su: HTML atributi, nazivi elemenata ili CSS klase.
  - TS kod za direktive definiše podatke šablona i ponašanje HTML elemenata.
- Angular kompajler pristupa DOM šablonu i kompajlira sve direktive, a zatim se povezuje sa opsegom važenja da bi bio kreiran novi prikaz u realnom vremenu.
- Prikaz u realnom vremenu sadrži elemente i funkcije DOM-a koji su definisani u direktivi.



Upotreba ugrađenih direktiva

- Tri kategorije *Angular* direktiva:
  1. Direktive komponenenata - direktive za rad sa šablonima
  2. Strukturalne direktive - direktive koje menjaju strukturu DOM (dodavanjem, uklanjanjem ili zamenom HTML elemenata u šablonu)
  3. Atributske direktive - direktive koje manipulišu izgledom i ponašanjem elemenata DOM-a
- Direktive komponenenata (@Component)
  – Komponente su „srce“ Angular tehnologije (osnovna gradivna jedinica), koja ima svoj šablon (HTML), stilove (CSS) i logiku (TypeScript fajl)
  – Komponentu kreira selektor koji se koristi kao HTML tag za dinamičko dodavanje (HTML + CSS + Angular logike) unutar DOM
  – Komponente se koriste za kreiranje modularnih i ponovno upotrebljivih delova korisničkog interfejsa, kroz menije, forme, kartice, itd.



Strukturalne direktive (1)

- **Ove direktive se obično koriste za kontrolu toga kako se elementi prikazuju na osnovu određenih uslova ili iteracija kroz kolekcije podataka.**
- Primeri strukturalnih direktiva uključuju: **ngIf, ngFor** i **ngSwitch.**
- **ngFor** - Koristi se za iteriranje kroz kolekcije podataka i generisanje HTML elemenata za svaku stavku u kolekciji. Primer:
```
<div *ngFor="let stavka of listastudenata"></div>
```
- **ngIf** – Koristi se za prikazivanje ili skrivanje elemenata u DOM-u na osnovu uslova. Dodajemo element u DOM ukoliko se vrati vrednost *true* (ako je *false*, taj element se uklanja iz DOM-a). Primer:
```
<div *ngIf="uslov"></div>
```
- **ngSwitch** - Omogućava prikazivanje jednog od više mogućih DOM elemenata, u zavisnosti od vrednosti izraza. Funkcioniše slično kao *ngIf*, odnosno ne kreira element, ako se njegova vrednost ne podudara sa nekom granom (*case*).

Strukturalne direktive (2)

- **ngSwitch** - Primer:

```
<div [ngSwitch]="timeOfDay">
    <span [ngSwitchCase]="'morning'">Jutro</span>
    <span [ngSwitchCase]="'afternoon'">Popodne</span>
    <span [ngSwitchDefault]="'daytime'">Uvece</span>
</div>
```

- **ngSwitchCase** - direktiva procenjuje uskladištenu vrednost i vrednost koja je prosleđena direktivi ngSwitch i utvrđuje da li je potrebno kreirati HTML šablon koji je pridružen.
- **ngSwitchDefault** - ako svi ngSwitchCase daje rezultat False

Kod ngFor i ngIf koristi se *, da bi se Angularu pokazala prisutnost directive u elementu.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <div *ngIf="condition">condition met</div>
    <div *ngIf="!condition">condition not met</div>
    <button (click)="changeCondition()">Change Condition</button> <hr>
    <template ngFor let-person [ngForOf]="people">
       <div>name: {{person}}</div>
    </template>
    <hr> <h3>Monsters and where they live</h3>
    <ul *ngFor="let monster of monsters">
       {{monster.name}}:{{monster.location}}
    </ul> <hr>
    <div [ngSwitch]="time">
     <span *ngSwitchCase="'night'">It's night time
        <button (click)="changeDay()">change to day</button>
     </span>
     <span *ngSwitchDefault>It's day time
     <button (click)="changeNight()">change to night</button></span>
    </div>
  `
})
```

```
export class AppComponent {
    condition: boolean = true;
    changeCondition = function(){
        this.condition = !this.condition;
    }
    changeDay = function(){
        this.time = 'day';
    }
    changeNight = function(){
         this.time = 'night'
    }
    people: string[] = ["Andrew", "Dillon", "Philipe", "Susan"]
    monsters = [
        { name: "Nessie", location: "Loch Ness, Scotland" },
        { name: "Bigfoot", location: "Pacific Northwest, USA" },
        { name: "Godzilla", location: "Tokyo, sometimes New York"
}
    ]
    time: string = 'night';
}
```

condition met

Change Condition

name: Andrew
name: Dillon
name: Philipe
name: Susan

Monsters and where they live

Nessie: Loch Ness, Scotland

Bigfoot: Pacific Northwest, USA

Godzilla: Tokyo, sometimes New York

Its night time change to day

Atributske direktive

- Atributske direktive su direktive koje se primenjuju na postojeće HTML elemente kako bi promenile ili proširile njihove osobine ili ponašanje.
- Za razliku od strukturalnih direktiva koje menjaju strukturu DOM-a, atributske direktive obično dodaju dodatne funkcionalnosti ili ponašanja postojećim elementima.

Direktiva	Opis
ngClass	Omogućava dinamičko dodavanje ili uklanjanje CSS klasa na osnovu uslova.
ngStyle	Omogućava dinamičko postavljanje stilova na osnovu vrednosti izraza.
ngModel	Ova direktiva omogućava vezivanje, prati promene u promenljivoj, a zatim ažurira vrednosti prikaza. Primer: `<input [(ngModel)]="text"> ` `<h1>{{text}}</h1>`

ngForm	Ova direktiva kreira grupu obrazaca i omogućava praćenje vrednosti i validaciju u okviru te grupe obrazaca. Pomoću direktive *ngSubmit*, možemo da podatke prosledimo događaju slanja kao objekat. Primer:
	`<form #formName="ngForm"`
	`(ngSubmit)="submitujFormu(formName)">`
	`</form>`

Primer atributskih direktiva (*TS fajl*)

```
import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   templateUrl: './attribute.component.html',
   styleUrls: ['./attribute.component.css']
})

export class AppComponent {
   colors: string[] = ["red", "blue",
                       "green", "yellow"];
   name: string;
   color: string = 'color';
   isDisabled: boolean = true;
   classes:string[] = ['bold', 'italic’,
                       'highlight'];
   selectedClass:string[] = [];
   enabler(){
      this.isDisabled = !this.isDisabled;
   }
```

```
   addClass(event: any){
        this.selectedClass = [];
        var values = event.target.options;
        var opt: any;

        for (var i=0, iLen = values.length; i<iLen; i++){
           opt = values[i];
            if (opt.selected){
               this.selectedClass.push(opt.text);
            }
        }
   }
}
```

Primer atributskih direktiva (*HTML fajl*)

```
<form>
   <span>name: </span>
   <input name="name" [(ngModel)]="name">
   <br> <span>color:</span>
   <input type="checkbox" (click)="enabler()">
   <select #optionColor [(ngModel)]="color" name="color" [disabled]="isDisabled">
      <option *ngFor="let color of colors" [value]="color">{{color}}</option>
   </select><hr>
   <span>Change Class</span><br>
   <select #classOption multiple name="styles" (change)="addClass($event)">
      <option *ngFor="let class of classes" [value]="class" >{{class}}</option>
   </select><br>
   <span>press and hold control/command <br>
   to select multiple options </span>
</form>
<hr>
<span>Name: {{name}}</span><br>
<span [ngClass]="selectedClass" [ngStyle]="{'color': optionColor.value}">
color: {{optionColor.value}}
</span><br>
```

Primer atributskih direktiva (*CSS fajl*)i izgled stranice

```
.bold {
   font-weight: bold;
}
.italic {
   font-style: italic;
}
.highlight {
   background-color: lightblue;
}
```

Prilagođene direktive (eng. *custom*)

- Omogućavaju da proširimo funkcionalnost HTML-a, tako što ćemo implementirati ponašanje elementa.
- Ako smo napisali kod koji će manipulisati DOM-om, trebalo bi da ga postavimo unutar prilagođene direktive.
- Prilagođena direktiva se implementira pomoću poziva klase @directive, na isti način na koji se definiše komponenta.
- Metapodaci klase @Directive treba da sadrže selektor direktive koji se koristi u HTML-u. Direktiva treba da bude smeštena u klasi izvoza Directive.

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

Primer - Zumiranje (1)

- Dodavanje prilagođene funkcionalnosti, sa ciljem da postignemo interakciju kod nekih elemenata.
- Direktiva zumiranja, kojom će se preko točkića miša uvećavati ili smanjivati slika.
- Osluškujemo točkić miša, i kada se on aktivira, u zavisnosti od smera pomeranja točkića i veličine promene, implementira se izmena elementa na koji primenjujemo opciju zumiranja.
- Uvoz klasa:
  - Directive, ElementRef, HostListener, Input, Renderer (sve iz paketa @angular/core)

Primer - Zumiranje (2)

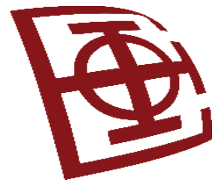
```
//zoom.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
   images: string[] = [
   '../assets/images/jump.jpg',
   '../assets/images/flower2.jpg',
   '../assets/images/cliff.jpg'
   ]
}
```

Primer - Zumiranje (3)

```
//zoom.directive.ts
import { Directive, ElementRef, HostListener, Input, Renderer } from '@angular/core';
@Directive({
   selector: '[zoom]'
})

export class ZoomDirective {
   constructor(private el: ElementRef, private renderer: Renderer) { }
   @HostListener('mouseenter') onMouseEnter() {
      this.border('lime', 'solid', '5px');
   }
   @HostListener('mouseleave') onMouseLeave() {
      this.border();
   }
   @HostListener('wheel', ['$event']) onWheel(event: any) {
      event.preventDefault();
      if(event.deltaY > 0){
         this.changeSize(-25);
      }
      if(event.deltaY < 0){
         this.changeSize(25);
      }
   }
```

Primer - Zumiranje (4)

```
//zoom.directive.ts nastavak
private border(
      color: string = null,
      type: string = null,
      width: string = null
   ){
      this.renderer.setElementStyle(
      this.el.nativeElement, 'border-color', color);
      this.renderer.setElementStyle(
      this.el.nativeElement, 'border-style', type);
      this.renderer.setElementStyle(
      this.el.nativeElement, 'border-width', width);
   }
   private changeSize(sizechange: any){
      let height: any = this.el.nativeElement.offsetHeight;
      let newHeight: any = height + sizechange;
      this.renderer.setElementStyle(
      this.el.nativeElement, 'height', newHeight + 'px');
   }
}
```



Primer - Zumiranje (5)

```
//app.component.html
<h1>Atributske direktive</h1>
<span *ngFor="let image of images">
   <img zoom src="{{image}}" />
</span>

//app.component.css
img {
   height: 200px;
}
```

Kreiranje prilagođene direktive pomoću komponente

- Angular komponente su vrsta direktiva
- Glavna razlika:
  komponente koriste HTML šablone za generisanje prikaza!
- Direktiva <ng-content> omogućava da preuzme postojeći HTML između 2 taga elementa u kojima se koristi direktiva i da upotrebi taj HTML u šablonu komponente.
- U sledećem primeru prikazano je korišćenje komponente,
  kao prilagođene direktive za promenu izgleda elementa,
  koji sadrži kontejnerski šablon.
- HTML šablon kontejnera elementa ima ulaze *title* i *description*,
  koji se koriste da bi se elementu dodali naslov i kratak opis.

Primer - Osnovna komponenta (1)

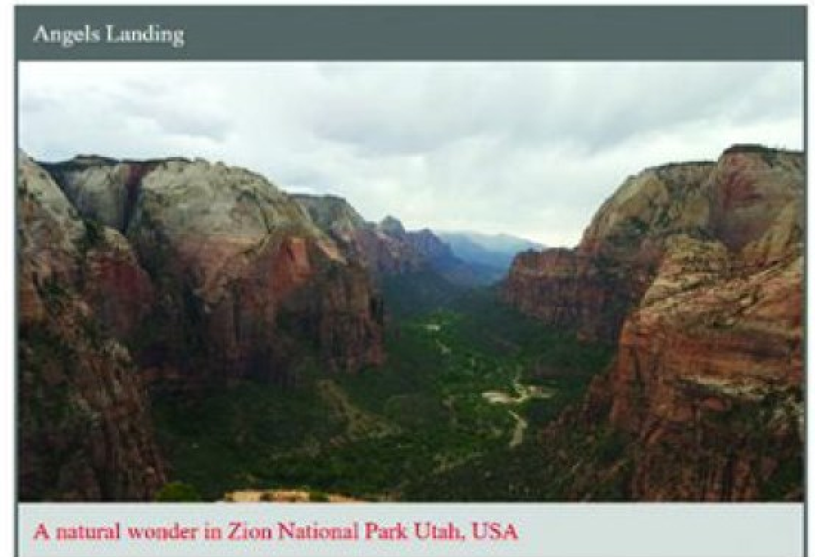
```
//Koreni fajl: app.component.ts

import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css'],
})
export class AppComponent {
images: any = [
{ src: "../assets/images/angelsLanding.jpg",
  title: "Angels Landing",
  description: "A natural wonder in Zion National Park Utah, USA" },
{ src: "../assets/images/pyramid.JPG",
  title: "Tikal",
  description: "Mayan Ruins, Tikal Guatemala" },
{ src: "../assets/images/sunset.JPG" },
]
}
```

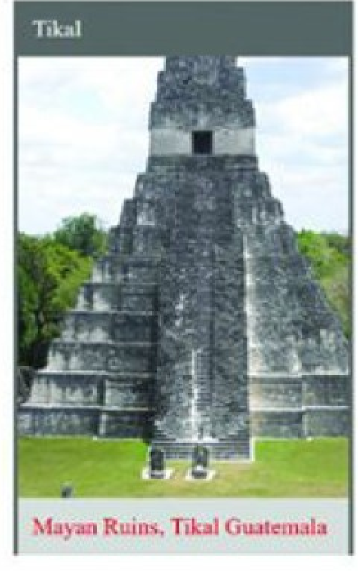
```
//HTML šablon kod osnovne komponente: app.component.html
<span *ngFor="let image of images" container title="{{image.title}}"
   description="{{image.description}}">
   <img src="{{image.src}}" />
</span>
<span container>
   <p>Lorem ipsum dolor sit amet</p>
</span>
<span container>
   <div class="diver">
   </div>
</span>

//CSS fajl, app.component.css
img { height: 300px; }
p { color: red }
.diver{
   background-color: forestgreen;
   height: 300px;
   width: 300px;
}
```

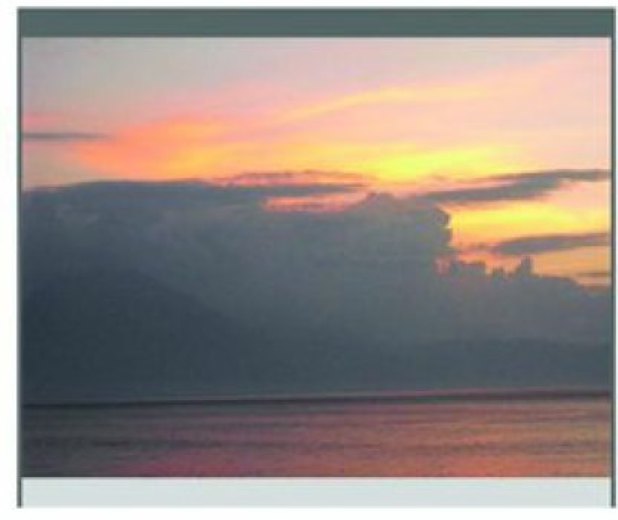
Primer - Komponenta kontejnera (3)

```
//TS kod za definisanje kontejnera - container.component.ts
import { Component, Input, Output } from '@angular/core';
@Component({
   selector: '[container]',
   templateUrl: './container.component.html',
   styleUrls: ['./container.component.css']
})
export class ContainerComponent {
   @Input() title: string;
   @Input() description: string;
}

//HTML kod za komponentu kontejnera
<div class="sticky">
   <div class="title"> {{ title }} </div>
   <div class="content">
      <ng-content></ng-content>
   </div>
   <div class="description"> {{ description }} </div>
</div>
```

```
.title {
   color: white;
   background-color: dimgrey;
   padding: 10px;
}
.content {
   text-align: center;
   margin: 0px;
}
.description {
   color: red;
   background-color: lightgray;
   margin-top: -4px;
   padding: 10px;
}
.sticky {
   display: inline-block;
   padding: 0px;
   margin: 15px;
   border-left: dimgrey 3px solid;
   border-right: dimgrey 3px solid;
}
```







Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore

Angular

Događaji i detekcija promena

Upotreba događaja pregledača

- Ugrađeni događaji funkcionišu kao povezivanje podataka
- Naziv događaja naveden u zagradama ( ) ukazuje Angularu na koji događaj treba da se povežemo

- Događaj je najčešće praćen pozivom neke funkcije, npr:

```
<input type="text" (change)="myEventHandler($event)"/>
```

HTML i Angular događaji

HTML sintaksa	Angular sintaksa	Opis događaja
onClick	(click)	Događaj koji se pokreće na klik na HTML element
onChange	(change)	Događaj koji se pokreće promenom vrednosti
onFocus	(focus)	Događaj koji se pokreće kada je izabran neki HTML element (stavljen fokus na njega)
onSubmit	(submit)	Događaj koji se pokreće kada se pošalje forma
onKeyUp, onKeyDown, onKeyPress	(keyup) (keydown) (keypress)	Događaji koji se pokreću kada se pritisnu tasteri na tastaturi
onMouseOver	(mouseover)	Događaji kojise pokreću kada se pokazivač miša nađe iznad nekog od HTML elemenata

Emitovanje prilagođenih događaja

- Događaji omogućavaju da pošaljete obaveštenje različitim nivoima u aplikaciji, kako biste im ukazali da su se pojavili događaji
- Primer: Ukazivanje podređenim komponentama da je promenjena vrednost u nadređenoj komponenti i obrnuto
- Klasa *EventEmitter* i metod *emit( )*, koji šalje događaj na najviše mesto u hijerarhiji nadređene komponente
- Sinktaksa:

```
@Output() name: EventEmitter<any> = new EventEmitter();
myFunction(){
    this.name.emit(args);
}
```

- name - naziv događaja, args - nula ili više argumenata koji se prosleđuju funkcijama rukovaoca događaja

Upravljanje prilagođenim događajima pomoću "osluškivača"

- Metod rukovaoca događaja koristi sintaksu u kojoj je name naziv događaja (iz prethodne tabele) koji se osluškuje,
  a *event* je vrednost koja je prosleđena klasi *EventEmitter*:

```
<div (name)="handlerMethod(event)">
```

Primer - Događaji u ugnežđenim komponentama

```
//customevent.component.ts - osnovna komponenta sa hendlerom dogadjaja
import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   templateUrl: 'customevent.component.html'
})
export class AppComponent {
   text: string = '';
   eventHandler(event: any){
      this.text = event;
   }
}
```

```
//customevent.component.html - HTML kod koji implementira prilagodjeni dogadjaj
<child (myCustomEvent)="eventHandler($event)"></child>
<hr *ngIf="text">
{{text}}
```

Primer - Podređena komponenta koja emituje događaj

```
//child.component.ts
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
   selector: 'child',
   template: `
   <button (click)="clicked()" (mouseleave)="mouseleave()">Click Me</button>
   `,
   styleUrls: ['child.component.css']
})
export class ChildComponent {
 private message = "";
 @Output() myCustomEvent: EventEmitter<any> = new EventEmitter();
 clicked() {
   this.message = "You've made a custom event";
   this.myCustomEvent.emit(this.message);
 }
 mouseleave(){
   this.message = "";
   this.myCustomEvent.emit(this.message);
 }
}
```

Primer - Podređena komponenta koja emituje događaj

CustomEvent

localhost:4200

Click Me

CustomEvent

localhost:4200

Click Me

You've made a custom event

Primer - Brisanje podataka u nadređenoj komponenti iz podređene

```
//character.component.ts
import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   character = null;
   characters = [ {name: 'Frodo', weapon: 'Sting', race: 'Hobbit'},
                  {name: 'Aragorn', weapon: 'Sword', race: 'Man'},
                  {name:'Legolas', weapon: 'Bow', race: 'Elf'},
                  {name: 'Gimli', weapon: 'Axe', race: 'Dwarf'} ]
   selectCharacter(character){ this.character = character; }
   deleteChar(event){
      var index = this.characters.indexOf(event);
      if(index > -1) {
         this.characters.splice(index, 1);
      }
      this.character = null;
   }
}
```

Primer - Brisanje podataka (2)

//character.component.html

```
<h2>Custom Events in Nested Components</h2>
<div *ngFor="let character of characters">
   <div class="char" (click)="selectCharacter(character)">
      {{character.name}}
   </div>
</div>
<app-character
   [character]="character"
   (CharacterDeleted)="deleteChar($event)">
</app-character>
```

Primer - Brisanje podataka (3)

```
//details.component.ts
import { Component, Output, Input, EventEmitter } from '@angular/core';
@Component({
   selector: 'app-character',
   templateUrl: './characters.component.html',
   styleUrls: ['./characters.component.css']
})
export class CharacterComponent {
 @Input('character') character: any;
 @Output() CharacterDeleted = new EventEmitter<any>();
 deleteChar(){
   this.CharacterDeleted.emit(this.character);
 }
}
```

Primer - Brisanje podataka (4)

```
//details.component.html
<div>
   <div *ngIf="character">
   <h2>Character Details</h2>
     <div class="cInfo">
      <b>Name: </b>{{character.name}}<br>
      <b>Race: </b>{{character.race}}<br>
      <b>Weapon: </b>{{character.weapon}}<br>
      <button (click)="deleteChar()">Delete</button>
     </div>
   </div>
</div>
```

CharacterSelect

localhost:4200

Custom Events in Nested Components

Frodo

Aragorn

Legolas

Gimli

CharacterSelect

localhost:4200

Custom Events in Nested Components

Frodo

Aragorn

Legolas

Gimli

Character Details

Name: Gimli
Race: Dwarf
Weapon: Axe
Delete

CharacterSelect

localhost:4200

Custom Events in Nested Components

Frodo

Aragorn

Legolas

Angular

Forme: reaktivne i vođene šablonom

Pristupi rukovanja korisničkim unosom

- **Reaktivne forme (*Reactive*)** ili **Forme vođene šablonom (*Template-driven*)**
- Obe hvataju događaje korisničkog ulaza iz Pogleda, validiraju unos korisnika, kreiraju modele forme i modele podataka za ažuriranje, i pružaju način praćenja promena.
- Oba pristupa se koriste i imaju svoje prednosti.

Reaktivne forme	Forme vođene šablonom
Omogućavaju direktan, eksplicitan pristup objektnom modelu forme.	Osloniti se na direktivne u šablonu da bismo kreirali osnovni objektni model i manipulisali njime.
Robusniji su: skalabilniji, imaju višekratnu upotrebu i testabilni su.	Lako se dodaju u aplikaciju, ali nisu skalabilni kao reaktivne forme.
Ukoliko su forme glavni deo vaše aplikacije, reaktivne su dobar izbor.	Korisne su za dodavanje jednostavne forme u aplikaciju, kao što je forma za registraciju na mejling liste (liste e-pošte). Ako imate veoma precizne zahteva za formama, forme vođene šablonom mogu da se uklope.

Ključne razlike u vrstama formi

	Reaktivne forme	Forme vođene šablonom
Podešavanje modela forme	Eksplicitno, kreira se u klasi komponente	Implicitno, kreira se od strane direktiva
Model podataka	Strukturirani i nepromenljivi	Nestrukturirani i promenljivi
Tok podataka	Sinhroni	Asinhroni
Validacija formi	Funkcije	Direktive

- **Reaktivne forme** su skalabilnije od **formi vođenih šablonom**.
- Reaktivne forme pružaju direktan pristup osnovnom API forme i koriste sinhroni tok podataka između Pogleda i Modela podataka, što olakšava kreiranje velikih formi.
- Reaktivne forme zahtevaju manje podešavanja za proces testiranja, a testiranje ne zahteva duboko razumevanje detektovanim promena da bi se pravilno testirala ažuriranja forme i validacija.
- Forme vođene šablonom se fokusiraju na jednostavne scenarije i nisu toliko upotrebljivi.
- Oni apstrahuju API forme i koriste asinhroni tok podataka između Pogleda i Modela podataka. Apstrakcija ovih formitakođe utiče na testiranje (testovi zahtevaju više podešavanja).

Postavljanje modela formi

- I reaktivne forme i forme vođene šablonom prate promene vrednosti između ulaznih elemenata forme sa kojima korisnici stupaju u interakciju (Pogled) i podataka iz forme u modelu komponente.
- Ova dva pristupa dele osnovne gradivne blokove, ali se razlikuju kako kreiraju i upravljaju uobičajenim instancama kontroli forme.

Osnovne klase	Detalji
FormControl	Prati vrednost i status validacije pojedinačnih kontrola forme.
FormGroup	Prati iste vrednosti i status za kolekciju kontrola forme.
FormArray	Prati iste vrednosti i status za niz kontrola forme.
ControlValueAccessor	Formira most između Angular FormControl intanci i gradivnih DOM elemenata.

Postavka u reaktivnim formama

REACTIVE FORMS

D

FORM CONTROL DIRECTIVE

Direct access to FormControl instance after link is created by FormControlDirective

Favorite Color:

Blue

FORM CONTROL INSTANCE

TEMPLATE-DRIVEN FORMS

Favorite Color:

Blue

D

NGMODEL DIRECTIVE

FORM CONTROL INSTANCE

Tok podataka u formama

- Kada aplikacija sadrži formu, Angular mora da zadrži pogled u sinhronizaciji sa modelom komponente i model komponente u sinhronizaciji sa pogledom.
- Kada korisnici često menjaju podatke u formi pogleda (V),
one moraju da se odraze u modelu podataka (M).
- Slično tome, kada programska logika menja vrednosti u modelu podataka (M), vrednosti moraju da se odraze u pogledu (V).
- Reaktivne forme i šablonski vođene forme se razlikuju po tome kako rukuju podacima koji potiču od korisnika ili iz programskih promena.
- Na narednim dijagramima prikazane su obe vrste toka podataka, za svaki tip forme.

Tok podataka u reaktivnim formama

REACTIVE FORMS - DATA FLOW (VIEW TO MODEL)

D
FORM CONTROL DIRECTIVE

input event fires

FormControl instance value is set and valueChanges event fires

favoriteColorControl.setValue("blue")

Favorite Color:
Blue

FORM CONTROL INSTANCE

{ }
TYPESCRIPT COMPONENT

1. Postavlja boju „Blue“ u polje.
2. Element forme emituje događaj.
3. Osluškuju se događaji na ulazu forme, i odmah se prenosi nova vrednost instanci *Form Control*.
4. *FormControl* emituje novu vrednost (opservabla).
5. Svi pretplatnici na *valueChanges* obs. dobijaju novu vrednost.

REACTIVE FORMS - DATA FLOW (MODEL TO VIEW)

D
FORM CONTROL DIRECTIVE

FormControl instance value is set and valueChanges event fires

favoriteColorControl.setValue("red")

view is updated

FORM CONTROL INSTANCE

{ }
TYPESCRIPT COMPONENT

Favorite Color:
Red

1. Korisnik poziva metodu favoriteColorControl.setValue(), koja ažurira *FormControl* vrednost.
2. *FormControl* emituje novu vrednost „Red“ (kroz opservablu).
3. Svi pretplatnici na *valueChanges* obs. dobijaju novu vrednost.
4. Ažurira se element novom vred.

Tok podataka u formama vođenim šablonom (1)

TEMPLATE-DRIVEN FORMS - DATA FLOW (VIEW TO MODEL)

Favorite Color: Blue → input event fires → NGMODEL DIRECTIVE, directive sets value → FORM CONTROL INSTANCE → ngModelChange event fires → { } TYPESCRIPT COMPONENT, this.favoriteColor = "blue"

PRVIH 5 KORAKA IDENTIČNO !

1. Postavlja boju „Blue“ u polje.
2. Element forme emituje događaj.
3. Osluškuju se događaji na ulazu forme, i odmah se prenosi nova vrednost instanci *Form Control*.
4. *FormControl* emituje novu vrednost (preko opservable).
5. Svi pretplatnici na *valueChanges* obs. dobijaju novu vrednost.
6. Poziva se i metoda **NgModel.viewToModelUpdate( )**, koji emituje *ngModelChange* događaj.

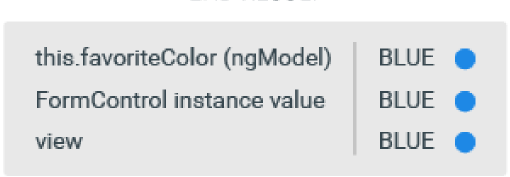
START

this.favoriteColor (ngModel)	RED
FormControl instance value	RED
view	BLUE

DIRECTIVE SETS VALUE

this.favoriteColor (ngModel)	RED
FormControl instance value	BLUE
view	BLUE

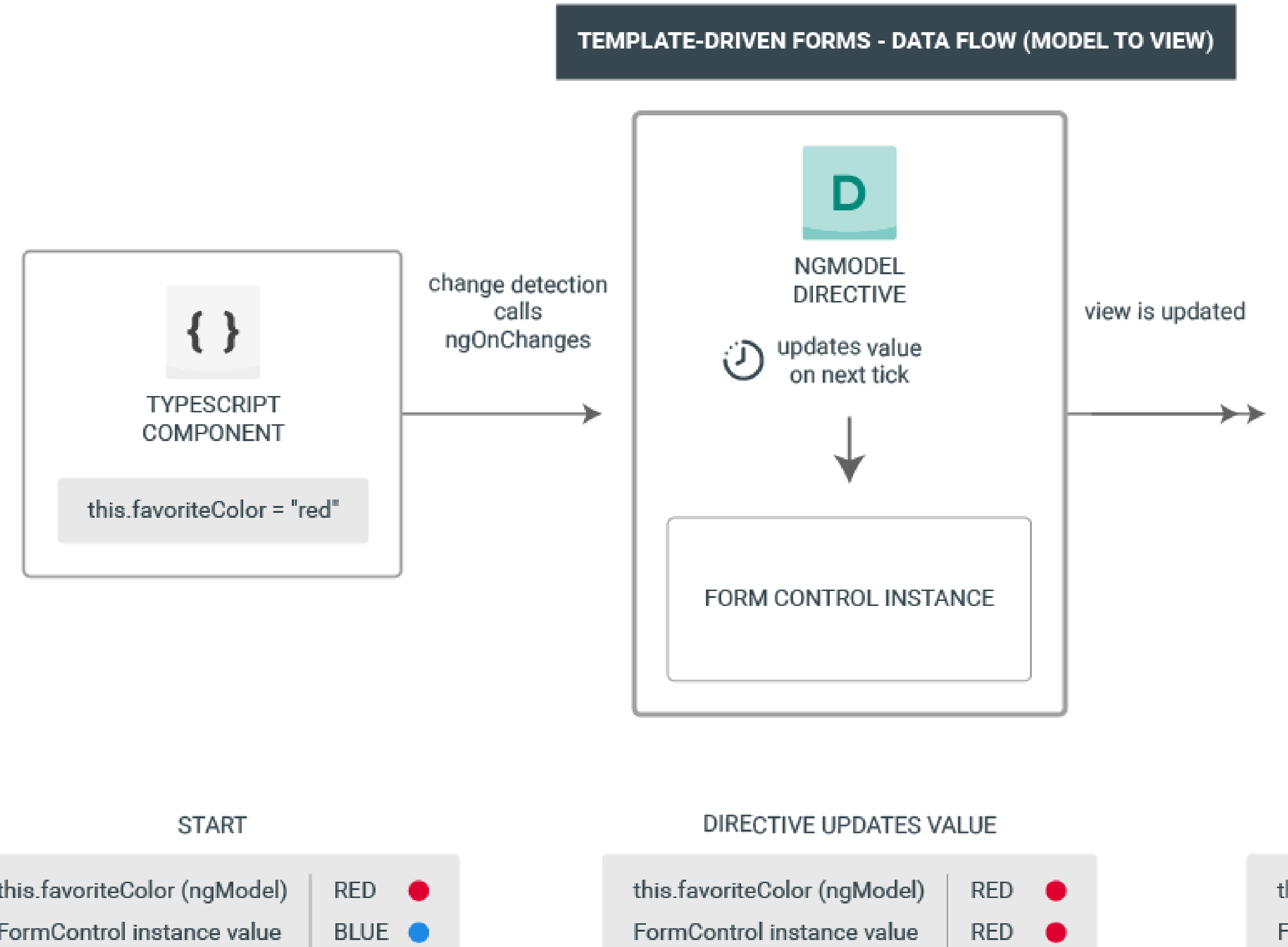
END RESULT



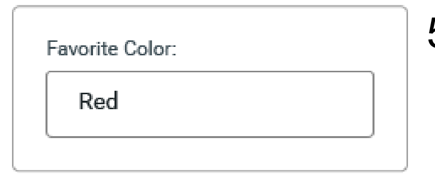
this.favoriteColor (ngModel)	BLUE
FormControl instance value	BLUE
view	BLUE

7. Kako komponenta šablona koristi dvostruko uvezivanje, svojstvo *favoriteColor* će biti ažurirano na vrednost emitovanu od *ngModelChange* događaja (**Blue**).

Tok podataka u formama vođenim šablonom (2)



1. Vrednost favoriteColor se ažurira u komponenti (M).
2. Detekcija promene počinje.
3. Tokom detekcije, *ngOnChanges* „udica“ je pozvana na *NgModel* direktivi instance (zato što se vred. jednog ulaza izmenila)
4. Metoda ngOnChanges postavlja async zadatak u red za postavljanje vrednosti za *FormControl* instancu.

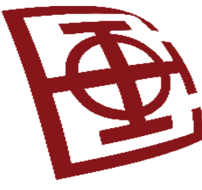


5. Detektovane promene kompletirane.
6. Na sledećem tiku, izvršava se zadatak postavljanja vred.
7. Instanca *FormControl* emituje poslednju vrednost kroz *valueChanges* opservablu.
8. Svi pretplatnici na *valueChanges* opservablu primaju novu vrednost i ažurira se input element u Pogledu.

Promenljivost modela podataka

Reaktivne forme	Forme vođene šablonom
Održavajte model podataka čistim (obezbediti ga kao nepromenljivu strukturu podataka).	Osloniti se na promenljivost sa dvosmernim uvezivanjem da biste ažurirali model podataka u komponenti, kako se promene vrše u šablonu.
Svaki put kada se promene promena na modelu podataka, FormControl instanca vraća novi model podataka (umesto da ažurira postojeći model). Pratite jedinstvene promene u modelu preko kontrolne opservable.	Pošto na modelu podataka nema jedinstvenih promena koje treba pratiti, kada se koristi dvosmerno vezivanje podataka, otkrivanje promena je manje efikasno u određivanju kada su ažuriranja potrebna.
Pošto ažuriranje podataka prati reaktivne obrasce, možete se integrisati sa operatorima opservable da biste transformisali podatke.	

- **Kod reaktivnih formi:** *FormControl* instanca uvek vraća novu vrednost kada se kontrolna vrednost ažurira.
- **Kod formi vođenih šablonom:** svojstvo omiljene boje se uvek menja na novu vrednost.



Validacija formi

- **Validacija** je sastavni deo upravljana bilo kojim skupom formi.
- Bez obzira da li se proveravaju obavezna polja (required) ili tražite eksterni API za postojeće korisničko ime, Angular obezbeđuje skup ugrađenih validatora, kao i mogućnost kreiranja specifičnih prilagođenih (custom) validatora.

Reaktivne forme	Forme vođene šablonom
Definišite prilagođene validatore kao funkcije koje dobijaju kontrolu za proveru.	Vezana je za direktive šablona i mora da obezbedi prilagođene direktive validatora, koje su omotač (wrap) validacionih funkcija.

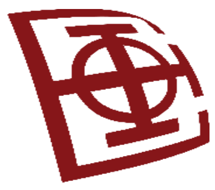
- Za više o samim validatorima pogledati: https://angular.io/guide/form-validation

Angular

Opservable u Angularu

Korišćenje opservabli kod tokova vrednosti

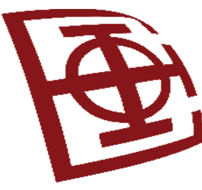
- **Opservable** je tehnika za rukovanje događajima, asinhrono programiranje i rukovanje višestrukim vrednostima emitovanim tokom vremena.
- Uzorak „*Observer*“ je softverski projektni uzorak u kome objekat, nazvan „*subject*“, održava listu svojih zavisnosti, nazvanih „*observers*“ (posmatrači) i automatski ih obaveštava o promenama stanja.
- Ovaj projektni uzorak je sličan (ali nije identičan) projektnom uzorku „publish/subscribe” (objavljivanje / pretplata).
- Angular aplikacije obično koriste RxJS biblioteku za opservable.



Opservable - terminologija

- **Opservable** su deklarativne. Programer definiše funkciju za objavljivanje vrednosti – izvor, ali ta funkcija se neće izvršiti sve dok se potrošač (*consumer*) ne pretplati na opserbavlu pozivanjem metode pretplate (*subscribe*).
- Ovaj pretplatnik zatim prima obaveštenje od opservable sve dok se ne završi, ili emituje greška ili potrošač ne otkaže pretplatu.
- Opservabla može da isporuči više vrednosti bilo kog tipa – literale, poruke ili događaje, u zavisnosti od konteksta. Tok pritiska na tastere, HTTP odgovor i otkucaji intervalskog tajmera su među najtipičnijim izvorima podataka. Opservable-in API se dosledno primenjuje na sve ove različite izvore.
- Opservabla može emitovati jednu, više ili nijednu vrednost, dok je pretplaćena. Može da emituje sinhrono (odmah emituje prvu vrednost) ili asinhrono (emituje vrednosti tokom vremena).
- Primer jednostavne opservable:

```
import { of } from 'rxjs';
const numbers$ = of(1, 2, 3); //emitovanje tri vrednosti sinhrono i zavrsetak
                              //Konvencija: oznaka $ na kraju naziva opservable
```



Pretplaćivanje (*subscribing*)

- **Opservabla** počinje da objavljuje vrednosti tek kada se neko pretplati na nju.
- Opservabla neće emitovati nikakve brojeve sve dok se neko ne pretplati, pozivanjem metode *subscribe( )*
- Primer :

```
numbers$.subscribe(
    value => console.log('Opservabla je emitovala sledece vrednosti:' + value)
);
```

- Proširenje funkcijom next( ) u subscribe( ) je pogodna sintaksa za ovaj tipičan slučaj.
- Opservabla ima tri tipa obaveštenja (notifikacije): *next, error* i *complete*.
- Observer (posmatrač) je objekat čija svojstva sadrže rukovaoce za ova obaveštenja:

Notifikacije	Opis
next	Rukovalac za svaku isporučenu vrednost. Poziva se nula ili više puta, nakon početka izvršavanja.
error	Rukovalac obaveštenja o grešci. Greška zaustavlja izvršavanje instance opservable i otkazuje pretplatu.
complete	Rukovalac o završetku izvršenja. Ne očekuje ponovni poziv next ili error. Automatski otkazuje pretplatu.

Subscribe metoda

- Primer kompletnog *Observer* objekta:

```
numbers$.subscribe({
    next: value => console.log('Opservabla emituje vrednost:' + value),
    error: err => console.log('Opservabla emituje gresku:' + err),
    complete: () => console.log('Opservabla emituje notifikaciju zavrsetka.')
});
```

- Ili kreiranjem *Observer* objekta sa funkcijama koje imenujemo kao *next( )*, *error( )*, i *complete( )*:

```
numbers$.subscribe({
    next(value) => console.log('Opservabla emituje vrednost:' + value),
    error(err) => console.log('Opservabla emituje gresku:' + err),
    complete() => console.log('Opservabla emituje notifikaciju zavrsetka.')
});
```

- Sva svojstva ovog rukovaoca su opciona!
- Ako izostavimo rukovalac za jedno od ovih svojstava, observer (posmatrač) će ignorisati obaveštenja tog tipa.

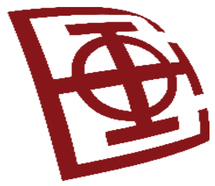
Upotreba opservabli

- **Opservable** obezbeđuju komponente za praćenje podataka, koji se asinhrono menjaju:
  - HTTP modul koristi observable da dohvati AJAX zahteve i odgovore;
  - Router i Forms moduli koriste observable da osluškuju podatke iz korisničkog unosa ili podatke ekoji pristižu sa servera.
- Cilj? Da možemo pratiti promene vrednosti.
- Vraćaju niz vrednosti; Niz ne mora biti primljen odjednom.
- Objekat *Opservable* se uvozi iz paketa rxjs/opservable, da bi mogao da se koristi u komponenti. RxJS operatori omogućavaju transformacije vrednosti opservabli.
- Nakon što se uveze, kreira se objekat:
  private name: Observable<Array<number>>
- Kada se objekat kreira, opservabla je dostupna za praćenje i ostatak komponente će moći da joj pristupi.
  Nakon implementacije, pratićemo je korišćenjem metode *subscribe*.

EventEmitter klasa

- **EventEmitter** klasa se koristi kada se objavljuju vrednosti iz komponente kroz dekorator @Output( )
- *EventEmitter* proširuje *RxJS Subject* dodajući metodu *emit( )*, tako da može slati proizvoljne vrednosti.
- Kada se pozove metoda *emit( )*, ona prosleđuje emitovanu vrednost do metode *next( )* svakog pretplaćenog posmatrača.
- Primer komponente koja osluškuje događaje otvaranja i zatvaranja:

```
<zippy (open)="onOpen($event)" (close)="onClose($event)">  </zippy>
```



Primer - *EventEmitter* (iz dokumentacije)

```
@Component({
  selector: 'zippy',
  template: `
  <div class="zippy">
    <div (click)="toggle()">
        Toggle
    </div>
    <div [hidden]="!visible">
      <ng-content></ng-content>
    </div>
 </div>`})
```

```
export class Zippy {
  visible: boolean = true;
  @Output() open: EventEmitter<any> = new EventEmitter();
  @Output() close: EventEmitter<any> = new EventEmitter();

  toggle() {
    this.visible = !this.visible;
    if (this.visible) {
      this.open.emit(null);
    } else {
      this.close.emit(null);
    }
  }
}
```

Primer - Objekat opservable

```
private name: Observable<Array<number>>;
ngOnInit(){
   this.name = new Observable(observer => {
     observer.next("my observable")
     observer.complete();
   }
   let subscribe = this.name.subscribe(
      data => { console.log(data) },
      Error => { errorHandler(Error) },
      () => { final() }
   );
  subscribe.unsubscribe();
}
```

=> prosleđivanje podataka opservabli

=> zatvaranje veze opservable

⇒ uspešan prijem podataka

⇒ hendler greške

⇒ završna funkcija (bez obzira na uspeh)

Primer - Opservable za detektovanje (1)

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/observable';
import { Subscription } from 'rxjs/Subscription';
@Component({
   selector: 'app-root',
   templateUrl: "./observable.component.html",
   styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
   combinedTotal:number = 0;
   private pass: Observable<any>;
   private run: Observable<any>;
   teams = [];
   ngOnInit(){
         this.teams.push({passing:0, running:0, total:0});
         this.teams.push({passing:0, running:0, total:0});
         //Passing
         this.pass = new Observable(observer => {this.playLoop(observer);});
         this.pass.subscribe(
           data => {
           this.teams[data.team].passing += data.yards;
           this.addTotal(data.team, data.yards);
         });
```

```
//Running
this.run = new Observable(observer => {
  this.playLoop(observer);
});
this.run.subscribe(
  data => {
  this.teams[data.team].running += data.yards;
  this.addTotal(data.team, data.yards);
});
//Combined
this.pass.subscribe(
  data => { this.combinedTotal += data.yards;
});
this.run.subscribe(
  data => { this.combinedTotal += data.yards;
});
}
```

Primer - Opservable za detektovanje (2)

```
playLoop(observer){
  var time = this.getRandom(500, 2000);
  setTimeout(() => {
      observer.next(
      { team: this.getRandom(0,2),
        yards: this.getRandom(0,30)});
        if(this.combinedTotal < 1000){
           this.playLoop(observer);
        }
      }, time);
  }
  addTotal(team, yards){
      this.teams[team].total += yards;
  }
  getRandom(min, max) {
     return Math.floor(Math.random() * (max - min)) + min;
  }
}
```

ŠABLON:

```
<div>
   Team 1 Yards:<br>
   Passing: {{teams[0].passing}}<br>
   Running: {{teams[0].running}}<br>
   Total: {{teams[0].total}}<br>
   <hr>
   Team 2 Yards:<br>
   Passing: {{teams[1].passing}}<br>
   Running: {{teams[1].running}}<br>
   Total: {{teams[1].total}}<hr>
   Combined Total: {{combinedTotal}}
</div>
```

Team 1 Yards:
Passing: 231
Running: 244
Total: 475

Team 2 Yards:
Passing: 196
Running: 138
Total: 334

Combined Total: 821

HTTPClient i observable

- Za razliku od *Promise*-a koji gura (push protokol) jednu vrednost, *Observable* može gurati više vrednosti tokom vremena
- Angular HTTPClient vraća opservable iz HTTP metoda
- Na primer: **`http.get('/api')`** vraća opservablu
- Ovo pruža nekoliko prednosti u odnosu na HTTP API zasnovane na obećanjima (*Promise*):
  - *Observable* ne menjaju odgovor servera (što se može dogoditi kod ulančanih .then( ) poziva kod *Promise*. Umesto toga, možete koristiti seriju operatora da transformišete vrednosti po potrebi.
  - HTTP zahtevi se mogu otkazati metodom *unsubscribe( )*
  - Zahtevi se mogu konfigurisati da dobijaju informaciju o napretku ažuriranja događaja
  - Neuspeli zahtevi se lako mogu ponoviti

Router i observable (1)

- *Router.events* pruža događaje kao opservable.
- Možete koristiti *filter( )* operator iz RxJS da tražite događaje od interesa i pretplatite se na njih kako biste donosili odluke na osnovu redosleda događaja u procesu navigacije.

```
import { Router, NavigationStart } from '@angular/router';
import { filter } from 'rxjs';
@Component({
  standalone: true,
  selector: 'app-routable',
  template: 'Routable1Component template'
})
export class Routable1Component implements OnInit {
  navStart: Observable<NavigationStart>;

  constructor(router: Router) {
    // Create a new Observable that publishes
    // only the NavigationStart event
    this.navStart = router.events.pipe(
      filter(evt => evt instanceof NavigationStart)
    ) as Observable<NavigationStart>;
  }
  ngOnInit() {
    this.navStart.subscribe(() =>
                          console.log('Navigation Started!'));
  }
}
```

Router i observable (2)

- *ActivatedRoute* je injektirani ruter servis koji koristi opservablu da bi dobila informacije o putanji rute i parametrima.
- Na primer: *ActivatedRoute.url* sadrži opservablu koja izveštava o putanji ili putanjama rute

```
import { ActivatedRoute } from '@angular/router';

@Component({
  standalone: true,
  selector: 'app-routable',
  template: 'Routable2Component template'
})
export class Routable2Component implements OnInit {
  constructor(private activatedRoute: ActivatedRoute) { }

  ngOnInit() {
    this.activatedRoute.url
      .subscribe(url => console.log('The URL changed to: '
                          + url));
  }
}
```

ReactiveForms i observable

- Reaktivne forme imaju svojstva koja koriste opserbable za praćenje kontrolnih vrednosti forme.
- Unutar klase FormControl, svojstva valueChanges i statusChanges sadrže opservable koje izazivaju događaje promena. Pretplaćivanje na opservablu form-control svojstva je način pokretanja logike aplikacije unutar klase komponente.

```
import { FormGroup } from '@angular/forms';
@Component({
  standalone: true,
  selector: 'app-hero-form',
  template: 'Hero Form Template'
})
export class HeroFormComponent implements OnInit {
  nameChangeLog: string[] = [];
  heroForm!: FormGroup;

  ngOnInit() {
    this.logNameChange();
  }
  logNameChange() {
    const nameControl = this.heroForm.get('name');
    nameControl?.valueChanges.forEach(
      (value: string) => this.nameChangeLog.push(value)
    );
  }
}
```

Angular

Implementacija veb servisa u veb aplikacijama

Angular servisi

- Svrha servisa je da obezbedi koncizan deo koda koji izvršava određene zadatke
- Servis može raditi nešto jednostavno ili nešto složeno;
  to je tipično klasa sa uskom, dobro definisanom svrhom.
- Servis obezbeđuje kontejner za ponovno upotrebljive funkcije koje su lako dostupne Angular aplikacijama
- Servisi su u Angularu definisani i registrovani pomoću mehanizma za injektiranje zavisnosti (*dependency injection*)
- Servisi se mogu pojaviti u modulima, u komponentama ili u drugim servisima
- Kada se na primer ugrade u modul, servisi se mogu koristiti u čitavoj aplikaciji

Razlike komponenti i servisa

- Komponente – za povećanje modularnosti i ponovne upotrebe
- Zadatak komponente je da omogući korisničko iskustvo i ništa više.
- Komponenta predstavlja svojstva i metode za povezivanje podataka, da posreduje između pogleda (renderovanih šablonom) i poslovne logike aplikacije (koja često uključuje neki model).
- Komponenta može delegirati određene zadatke servisima, kao što su:
  - preuzimanje podataka sa servera,
  - validacija korisničkog unosa,
  - izveštavanje (log) direktno u konzolu.
- Definisanjem takvih zadataka obrade u servisu koji se može injektirati, čini te zadatke dostupnim bilo kojoj komponenti
- Servisi su dostupni komponentama kroz injektiranu zavisnost (DI)!

Primer jednostavnih klasa kao servisa

- Servis koji izveštava (loguje) sadržaj na konzoli veb pregledača →

```
//Fajl: src/app/logger.service.ts

export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

- Servis zasnovan na drugim servisima:

```
export class HeroService {
  private heroes: Hero[] = [];
  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

Kreiranje servisa

- Pokretanjem komande u Angular CLI: **ng generate service heroes/hero**
- Ovom komandom biće napravljen podrazumevani servis:

```
import { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';  //mokovani podaci
@Injectable({
    //deklarise da ovaj servis treba da bude kreiran od strane root injektora
    providedIn: 'root',
})
export class HeroService {
    getHeroes() { return HEROES; }
}
```

- Dekorator **@Injectable( )** specificira da Angular može koristiti tu klasu u DI sistemu.
- Metapodaci **providedIn: 'root'** znači da je ovaj servis vidljiv kroz aplikaciju.
- Korišćenje servisa u komponenti, npr. kao zavisnost u konstruktoru/argument:

```
constructor(heroServis: HeroService)
```

Injektiranje servisa u druge servise

- Slično kao kod injektiranja u komponentu.
- Primer uključivanja *Logger* servisa u *HeroService* i njegov konstruktor:

```
//Fajl: src/app/heroes/hero.service

import { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';
import { Logger } from '../logger.service';

@Injectable({
  providedIn: 'root',
})
export class HeroService {

  constructor(private logger: Logger) {  }

  getHeroes() {
    this.logger.log('Getting heroes ...');
    return HEROES;
  }
}
```

```
//Fajl: src/app/logger.service

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

@Injectable({
  providedIn: 'root'
})
export class Logger {
  logs: string[] = [];
  // capture logs for testing

  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

Servisi ugrađeni u Angular

Naziv servisa	Opis
animate	Obezbeđuje animirane "kukice" (*hooks*) za povezivanje sa *CSS* i *JavaScript* animacijama
http	Obezbeđuje jednostavne funkcije za slanje HTTP zahteva na veb server ili druge servise
router	Obezbeđuje pristup prikazima i odeljcima u okviru prikaza
forms	Obezbeđuje servis koji omogućava dinamičke i reaktivne forme sa jednostavnim obrascem validacije

Slanje HTTP zahteva pomoću http servisa

- Servis http omogućava da direktno komunicirate sa veb serverom iz Angular koda (servis u osnovi koristi *XMLHttpRequest*)
- Možemo zahtevati podatke sa servera (npr. GET) ili slati podatke na server (npr. PUT/POST)
- Postoje dva načina upotrebe http servisa. Najjednostavniji je upotreba ugrađenih skraćenih metoda koji odgovaraju standardnim HTTP zahtevima:
  - delete (url, [options])
  - get (url, [options])
  - head (url, [options])
  - post (url, data, [options])
  - put (url, data, [options])
  - patch (url, data, [options])

url = URL veb zahteva; data = podaci; options = JS objekat koji određuje opcije;

Svojstva (*options*) kod http zahteva

Svojstvo	Opis
headers	Zaglavlja koja se šalju u zahtevu. Možete da odredite objekat koji sadrži nazive zaglavlja koji se šalju kao svojstva. Ako svojstvo u objektu ima vrednost null, zaglavlje neće biti poslato.
observe	body ili events ili response (specificira kakav odgovor treba da se vrati)
params	Parametri koji se šalju (ili u JSON stringu, ili broj ili Bulova ili standardni string nekog oblika: ?kljuc1=vred1&kljuc2=vred2&...)
reportProgress	Bulova vrednost (da biste slušali napredak događaja prilikom prenosa većih količina podataka)
responseType	Željeni tip odgovora (vraćenih podataka) koji se očekuje: arraybuffer ili blob ili json ili text
withCredentials	Kada je vrednost flega true, znači da je fleg postavljen na objekat XHR

Pokretanje HTTP zahteva

- Za sve HttpClient metode, metod neće otpočeti HTTP zahtev dok ne pozovete *subscribe( )* na opservabli.
- Pozivanjem *subscribe( )* pokreće se izvršavanje opservable i uzrokuje da HttpClient kreira i pošalje HTTP zahtev serveru.
- Primer:

```
const req = http.get<Heroes>('/api/heroes');
// 0 requests made - .subscribe() not called.
req.subscribe();
// 1 request made.
req.subscribe();
// 2 requests made.
```

Zahtevanje tipiziranog odgovora

- Strukturirajte svoj HttpClient zahtev da deklariše tip objekta odgovora.
- Loše rešenje: (povratni tip Objekat)

```
return this.http.get(this.configUrl);
```

- Zašto je loše?
  Da bi pristupali svojstvima, neophodna je eksplicitna konverzija sa „as any“:

```
showConfig() {
  this.configService.getConfig()
    .subscribe(data => this.config = {
        heroesUrl: (data as any).heroesUrl,
        textfile:  (data as any).textfile,
        date: (data as any).date,
    });
  }
```

Sigurnije je ukoliko vraćeni objekat ima željeni tip!

Zahtevanje tipiziranog odgovora (2)

- Definisati interfejs sa potrebnim svojstvima umesto klase,
  jer je odgovor običan (plain) objekat koji se ne može automatski konvertovati u instancu klase:

```
export interface Config {
  heroesUrl: string;
  textfile: string;
  date: any;
}
```

- Sada navodimo taj interfejs kao HttpClient.get( ) poziv u servisu:

```
getConfig() {
  // sada je povratni tip Observable od interfejsa Config
  return this.http.get<Config>(this.configUrl);
}
```

Zahtevanje tipiziranog odgovora (3)

- Prijem objekta u komponenti Config Component, sa tipiziranim odgovorom:

```
config: Config | undefined;
showConfig() {
  this.configService.getConfig()
    .subscribe(data => this.config = {
        heroesUrl: data.heroesUrl,
        textfile:  data.textfile,
        date: data.date,
    });
}
```

... ili sa destrukturiranim dodeljivanjem:

```
config: Config | undefined;

showConfig() {
  this.configService.getConfig()
    // clone the data object,
    // using its known Config shape
    .subscribe(data => this.config = { ...data });
}
```

Čitanje punog odgovora

- Pun odgovor sa opservablom:

```
getConfigResponse(): Observable<HttpResponse<Config>> {
  return this.http.get<Config>(
    this.configUrl, { observe: 'response' });
}
```

- Metoda komponente *showConfigResponse( )* prikazuje zaglavlje odgovora:

resp je tipa`HttpResponse<Config>`

```
showConfigResponse() {
  this.configService.getConfigResponse().subscribe(resp => {
      const keys = resp.headers.keys();
      this.headers = keys.map(key =>
        `${key}: ${resp.headers.get(key)}`);
      // pristup telu direktno, tipiziran je kao `Config`.
      this.config = { ...resp.body! };
    });
}
```

Implementiranje funkcija povratnog poziva HTTP odgovora

- Kada pozovete metod zahteva pomoću objekta http, dobićete objekat *Observable*, koji omogućava neprekidno praćenje poslatih ili primljenih podataka na server.
- Neki korisni metodi:
  - *map* - Primenjuje funkciju na svaku vrednost u sekvenci opservable. Ovo omogućava da dinamički transformišete izlaz toka opservable u prilagođene formate podataka.
  - *toPromise* - Konvertuje opservablu u objekat Promise, koji pristupa metodama u promisu. Objekti Promise obezbeđuju sintaksu za upravljanje asinhronim operacijama.
  - *catch* - Određuje funkciju za lepo upravljanje greškama u sekvenci opservable.
  - *debounce* - Obezbeđuje interval u kome će tok opservable emitovati vrednost. Biće emitovana samo vrednost u intervalu, dok međuvrednosti neće biti emitovane.

Pravljenje DELETE zahteva i PUT zahteva

- Primer:
```
/** DELETE: Brisanje heroja sa servera */
deleteHero(id: number): Observable<unknown> {
    const url = `${this.heroesUrl}/${id}`; // DELETE api/heroes/42
    return this.http.delete(url, httpOptions)
    .pipe(
      catchError(this.handleError('deleteHero'))
    );
}
```

Poziv:
```
this.heroesService
   .deleteHero(hero.id)
   .subscribe();
```

- Primer
```
/** PUT: ažuriranje heroja na serveru. Vraća ažuriranog heroja u uspešnom. */
updateHero(hero: Hero): Observable<Hero> {
    return this.http.put<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('updateHero', hero))
    );
}
```

Implementiranje JSON datoteke i upotreba http servisa za pristup

```json
[
{
    "userId": 1,
    "userName": "brendan",
    "userEmail": "fake@email.com"
},
{
    "userId": 2,
    "userName": "brad",
    "userEmail": "email@notreal.com"
},
{
    "userId": 3,
    "userName": "caleb",
    "userEmail": "dummy@email.com"
},
{
    "userId": 4,
    "userName": "john",
    "userEmail": "ridiculous@email.com"
},
{
    "userId": 5,
    "userName": "doe",
    "userEmail": "some@email.com"
}
]
```

Komponenta koja implementira servis za GET zahtev

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { Http } from '@angular/http';
import 'rxjs/Rx';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.CSS']
})
export class AppComponent {
   users = [];
   constructor(private http: Http){
        http.get('../assets/dummyDB.JSON')
           .toPromise()
           .then((data) => {
             this.users = data.JSON()
           })
           .catch((err) =>{ console.log(err);})
   }
}
```

Modul koji uvozi HttpModule

```
//app.module.ts
import { BrowserModule } from '@angular/platform-
browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpModule } from '@angular/http';
import { AppComponent } from './app.component';

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

Lista korisnika isčitana iz baze (JSON)

```
//http.component.html
<h1>Korisnici</h1>
<div class="user" *ngFor="let user of users">
   <div><span>Id:</span> {{user.userId}}</div>
   <div><span>Username:</span> {{user.userName}}</div>
   <div><span>Email:</span> {{user.userEmail}}</div>
</div>
```

Users

Id: 1
Username: brendan
Email: fake@email.com

Id: 2
Username: brad
Email: email@notreal.com

Id: 3
Username: caleb
Email: dummy@email.com

Id: 4
Username: john
Email: ridiculous@email.com

Id: 5
Username: doe
Email: some@email.com

Angular

Arhitektura aplikacije sa *Spring Boot* ili *Node.JS*

Arhitektura SpringBoot aplikacije na serverskoj strani i povezivanje sa klijentskim delom

Korisnik

Klijentski deo

Serverski deo

Veb aplikacija / Mobilna aplikacija

iOS

TS

spring boot

Spring Rest Controller

Spring Data JPA

MySQL

PostgreSQL

Arhitektura Node.JS aplikacija na serverskoj strani i povezivanje sa klijentskim delom

Korisnik

Klijentski deo

Serverski deo

Veb aplikacija / Mobilna aplikacija

iOS

ili

React JS

ES6

TS

Express JS

node JS

mongoose

Sequelize

redis

MySQL

PostgreSQL

mongoDB

MEAN puni stek

2 Parse Request

1 Make Request

3 Get Database

Client

Angular

NodeJS Server

ExpressJS

Mongo DB

6 Display Response

4 Return Database

5 Return Request

Hvala na pažnji ☺

PITANJA?