Uvod v Angular

Boris Ovčjak

boris.ovcjak@gmail.com

boris.ovcjak@um.si

Gregor Jošt

jost.gregor@gmail.com

gregor.jost@um.si

UM FERI, Inštitut za informatiko

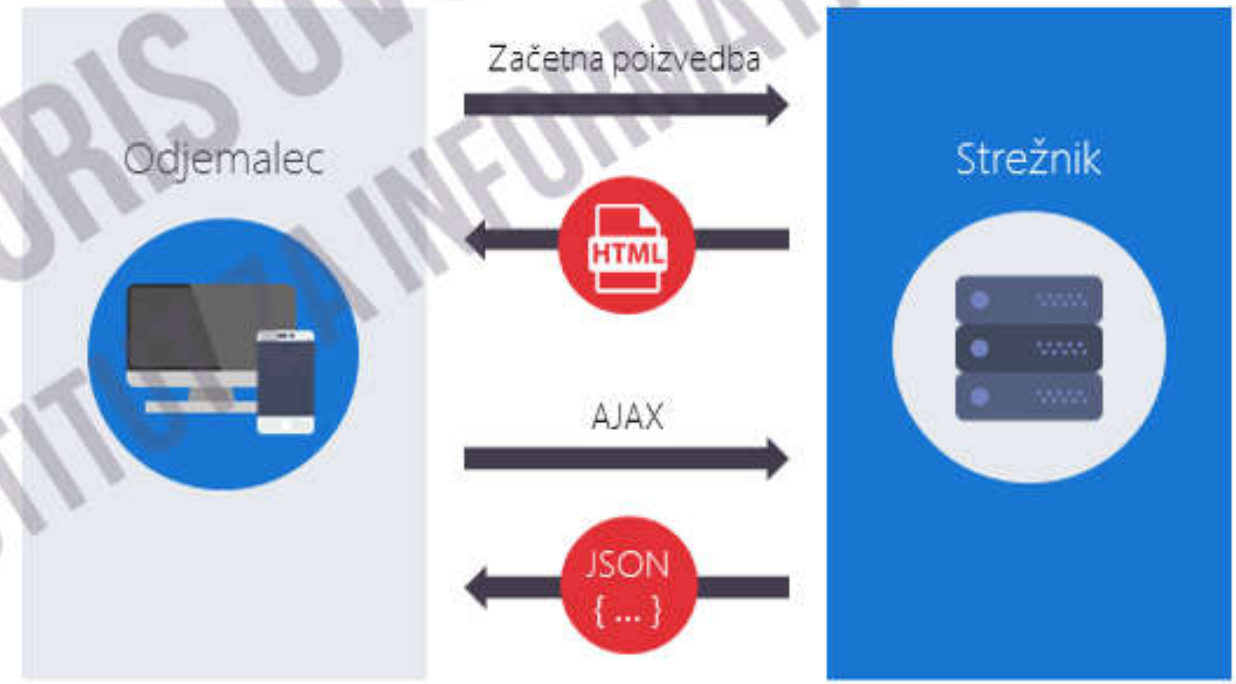
PREDEN ZAČNEMO

Povzetki prosojnic bodo na voljo na spletni strani konference

UVOD V ANGULAR

Angular je JavaScript ogrodje za razvoj odjemalskih (angl. Front-end) spletnih aplikacij

- Namenjen razvoju odjemalskih aplikacij,
- Trenutna različica: **4.2.2**
- Uporabljen na preko **12.000 spletnih straneh**.



UVOD V ANGULAR

lumx

ionic

Supersonic

MEAN.IO

Mobile Angular UI

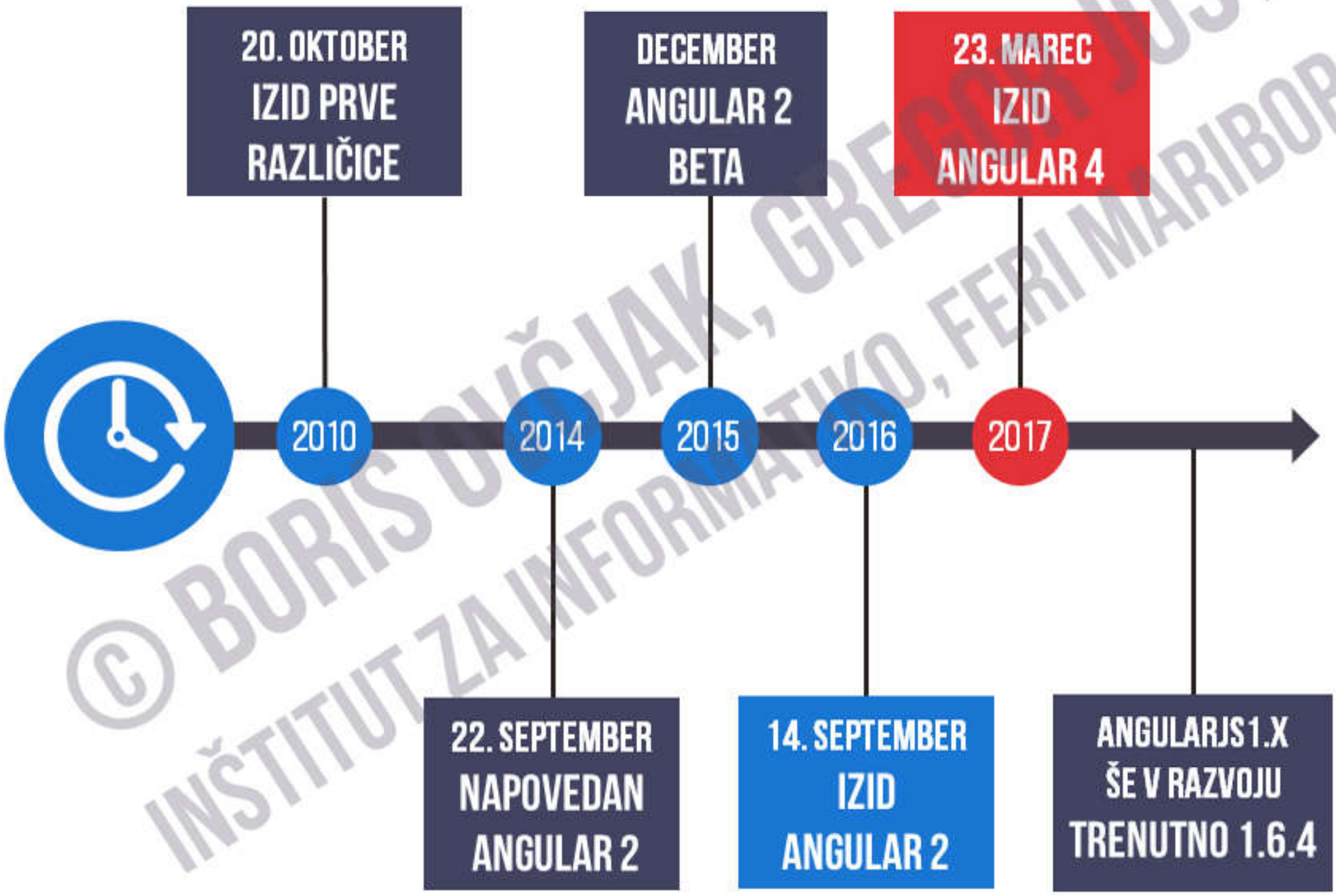
rad

Protractor

end to end testing for AngularJS

UI Bootstrap

ZGODOVINA OGRODJA ANGULAR



ZNAČILNOSTI ANGULAR

- Angular ni nastal kot nova verzija, temveč popolnoma novo ogrodje.

GLAVNE ZNAČILNOSTINOVEGA OGRODJA

- **Mobilni razvoj** – Fokusirana razvoj mobilnih aplikacij,
- **Modularnost** – veliko modulov je bilo odstranjenih z jedra Angular, kar rezultira v boljši zmogljivosti
- **Modern:**
  - Ciljana ES6, kar pomeni, da je na voljo za modern brskalnike in s tem zmanjša skrb za podporo starejšim brskalnikom preko različnih prilagoditev,
  - Podpira prednosti ES6
- Priporoča uporabo skriptnega **jezika TypeScript** (nadgradnja ES6), ki omogoča:
  - Objektno programiranje na osnovi razredov,
  - Tipiziranje,
  - Generike,
  - Lambda izraze

ZNAČILNOSTI ANGULAR

GLAVNE ZNAČILNOSTI NOVEGA OGRODJA

- Izboljšana tehnika vrivanja odvisnosti (angl. Dependency injection),
- Dinamično nalaganje,
- Asinhrono prevajanje predlog,
- Preprostejša navigacija,
- Knjižnica za beleženje (Diary.js),
- Menjava krmilnikov (angl. controller) in $scope s komponentami in direktivami,
- Reaktivno programiranje s podporo RxJS

PREDNOSTI IN SLABOSTI

Namenjen aplikacijam z veliko interaktivne odjemalske kode

Dobra rešitev za dinamične enostranske aplikacije

Hiter proces razvoja

V nekaterih primerih zahteva manj kode

Napredne funkcionalnosti za testiranje

Zmogljivost

Podpora spletnim komponentam

Uravnoteženost med modelom, pogledom in krmilnikom

Uporaba programskega jezika TypeScript

Orodje za poenostavitev izgradnje aplikacij

PREDNOSTI IN SLABOSTI

Počasen pri prikazovanju velikih količin podatkov. Razlog: neposredna manipulacija DOM

Nekoliko neprijazen glede optimizacije za spletne iskalnike - SEO (search engine optimization)

Težave pri učenju novih konceptov.

Učenje novega, bolj objektnega pristopa k JS. TS kodo je potrebno prevajati

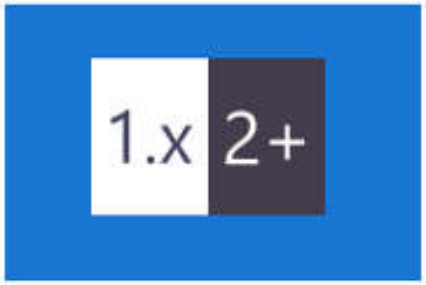
Težave pri nadgradnji iz ogrodja AngularJS

Izboljšava na račun **Angular Universal** – projekt, ki omogoča izvajanje Angular aplikacije na strežniku.

PODPORA OGRODJA

Chrome	Firefox	Edge	Internet Explorer	Safari	macOS	Android	Windows
Zadnji	Zadnji	14	11	10	10	6.0	11
		13	10	9	9	5.x	
			9	8	8	4.4	
				7	7	4.1-4.3	

VSEBINA



PRIMERJAVA

ARHITEKTURA

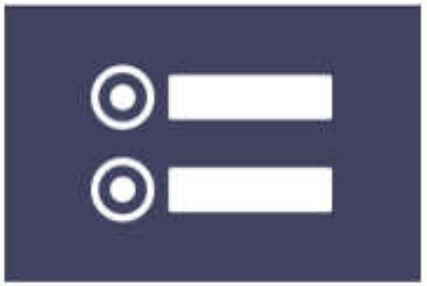
PRVA APLIKACIJA

ROOT MODUL

TYPESCRIPT

DELO S PODATKI

DIREKTIVE



RAZVOJ OBRAZCEV

NAVIGACIJA

STORITVE

HTTP

ANGULAR 4.X

PRIMERJAVA ANGULAR IN ANGULARJS

PRIMERJAVA ANGULAR IN ANGULARJS

OSNOVE PREDLOG

Povezovanje / interpolacija (angl. Bindings / Interpolation)

```
Ime osebe: {{vm.ime}}
```

- Vrednost elementa -> lastnost v krmilniku predloge („controller as" s predpono (vir)).

```
Ime osebe: {{ime}}
```

- Referenčna spremenljivka tukaj ni potrebna, saj vedno gre za asociirano komponento.

Filtri → Pipe

```
<td>{{oseba.ime | uppercase}}</td>
```

- Za filtriranje izhoda se uporablja znak pipa (|), ki mu sledi eden ali več **filtrov**.

```
<td>{{oseba.ime | uppercase}}</td>
```

- Podobna sintaksa z znakom pipa (|), vendar se sedaj tudi koncept imenuje **pipa**.

Lokalne spremenljivke → Vnosne spremenljivke

```
<tr ng-repeat="oseba in vm.osebe">
  <td>{{oseba.ime}}</td>
</tr>
```

- **oseba** je v tem primeru uporabniško definirana lokalna spremenljivka

```
<tr *ngFor="let oseba of osebe">
  <td>{{oseba.ime}}</td>
</tr>
```

- Uporaba vnosne spremenljivke predloge, ki je eksplicitno definirana z **let**

ANGULARJS | ANGULAR

PRIMERJAVA ANGULAR IN ANGULARJS

DIREKTIVE PREDLOG

ng-app

```
<body ng-app="sifrantOseb">
```

- Proces zagona aplikacije se imenuje **bootstrapping**.
- Čeprav lahko zaženemo vsako AngularJS aplikacijo preko kode, se večina aplikacij zažene z direktivo **ng-app**(ki ji podamo ime aplikacijskega modula (sifrantOseb)).
- Angular ne vključuje direktive za postavitev. Za zagon aplikacije je potrebno postaviti aplikacijski korenski modul (v kodi)

nalaganje -Bootstrapping

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

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

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

import { AppComponent }  from './app.component';

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

ANGULARJS

ANGULAR

PRIMERJAVA ANGULAR IN ANGULARJS

DIREKTIVE PREDLOG

ng-controller → Komponenta

```
<div ng-controller="MovieListCtrl as vm">
```

- Direktiva **ng-controller** skrbi za določanje krmilnika pogledu.

```
@Component({
  selector: 'movie-list',
  templateUrl: './movie-list.component.html',
  styleUrls: ['./movie-list.component.css' ],
})
```

- V novi verziji se krmilnik ne določa v predlogi, temveč za to skrbi komponenta, ki določi asociirano predlogo kot del razrednega dekoratorja komponente.

ng-if → *ngIf

```
<table ng-if="movies.length">
```

- Direktiva **ng-if** odstrani ali ponovno kreira del DOM-a glede na rezultat izraza.

```
<table *ngIf="movies.length">
```

- V novi verziji *ngIf deluje enako.
- * = **syntactic sugar** (označuje kompleksnejši izraz)

ANGULARJS

ANGULAR

PRIMERJAVA ANGULAR IN ANGULARJS

DIREKTIVE PREDLOG

ng-click → Povezovanje dogodka *click*

```
<button ng-click="vm.toggleImage()">
<button ng-click="vm.toggleImage($event)">
```

- **ng-click** direktiva omogoča določanje poljubnega obnašanja ob kliku na element
- Lahko pošljemo tudi objekt dogodka ($event)

```
<button (click)="toggleImage()">
<button (click)="toggleImage($event)">
```

- V novi verziji direktive vezane na dogodke ne obstajajo. Nadomesti jih enosmerna povezava iz predloge do komponente s povezovanjem dogodkov (angl. **event binding**)

ng-model → ngModel

```
<input ng-model="vm.favoriteHero"/>
```

- Direktiva **ng-model** poveže vnosno polje obrazca z lastnostjo v krmilniku.
- Dvosmerna povezava.

```
<input [(ngModel)]="favoriteHero" />
```

- V novi verziji se dvosmerna povezava določi z **[()]** ali „banana in a box" – kratica za definicijo tako povezovanja lastnosti kot dogodkov

ANGULARJS

ANGULAR

PRIMERJAVA ANGULAR IN ANGULARJS

DIREKTIVE PREDLOG

ng-show / ng-hide → Določanje lastnosti *hidden*

```
<h3 ng-show="vm.movie">
  Your favorite movie is: {{vm.movie.title}}
</h3>
```

- Direktiva **ng-show / ng-hide** prikaže oz. skrije asociiran HTML element glede na določen izraz.

```
<h3 [hidden]="!movie">
  Your favorite movie is: {{movie.title}}
</h3>
```

- V novi verziji za to skrbi določanje lastnosti; ogrodje ne vsebuje namenske vgrajene direktive, zato uporabljamo HTML lastnost *hidden*.

ng-repeat → *ngFor

```
<tr ng-repeat="movie in vm.movies">
```

- Direktiva **ng-repeat** ponavlja asociiran DOM element za vsak element v kolekciji.

```
<tr *ngFor="let movie of movies">
```

- ***ngFor** se obnaša podobno kot ng-repeat.
- Definiran element (tr) in njegovo vsebino spremeni v predlogo, in le-to uporablja za instanciiranje pogleda za vsak element v seznamu.

ANGULARJS

ANGULAR

PRIMERJAVA ANGULAR IN ANGULARJS

MODULI / KRMILNIKI/ KOMPONENTE

Angular moduli

```
angular.module("movieHunter", ["ngRoute"]);
```

- Angular modul skrbi za pregled nad krmilniki, storitvami in drugo kodo.
- Drugi argument definira seznam drugih modulov, od katerih je ta modul odvisen

Angular moduli

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

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

- V novi verziji je modul definiran z NgModule dekoratorjem in določa:
  - **Vključitve (imports)** – seznam modulov od katerih je odvisen
  - **Deklaracije (declarations)** – skrbi za pregled nad seznamom komponent, pip in direktiv

ANGULARJS

ANGULAR

PRIMERJAVA ANGULAR IN ANGULARJS

MODULI / KRMILNIKI/ KOMPONENTE

Registracija krmilnika

```
angular
  .module("movieHunter")
  .controller("MovieListCtrl",
              ["movieService",
               MovieListCtrl]);
```

- AngularJS vsebuje kodo v vsakem krmilniku, ki išče primeren Angular modul in pri njem registrira ta krmilnik.
- Prvi argument je ime krmilnika, sledi mu niz imen vseh odvisnosti, ki bodo vrinjene v ta krmilnik, ter referenca do funkcije krmilnika.

Dekorator komponente

```
@Component({
  selector: 'movie-list',
  templateUrl:'./movie-list.component.html',
  styleUrls: ['./movie-list.component.css'],
})
```

- V novi verziji Angular doda dekorator razredu komponente z namenom dodajanja potrebnih metapodatkov.
- Dekorator **@Component** deklarira, da je ta razred komponenta in določi metapodatke komponente, kot je selektor, predloga, ...

ANGULARJS

ANGULAR

PRIMERJAVA ANGULAR IN ANGULARJS

MODULI / KRMILNIKI/ KOMPONENTE

Funkcija krmilnika

```
function MovieListCtrl(movieService) {}
```

- V AngularJS se programska koda za model in metode pišejo v funkcijo krmilnika.

Razred komponente

```
export class MovieListComponent {}
```

- V Angular kreiramo razred komponente.

Injeciranje odvisnosti

```
MovieListCtrl.$inject = ['MovieService'];
function MovieListCtrl(movieService) {
}
```

- V AngularJS se vse odvisnosti podajo krmilniku kot funkcijski argumenti.
- Za varovanje pred problemi minifikacije je potrebno Angularju eksplicitno povedati, da mora vriniti instanco storitve MovieService v prvi parameter.

Injeciranje odvisnosti

```
constructor(movieService: MovieService) {
}
```

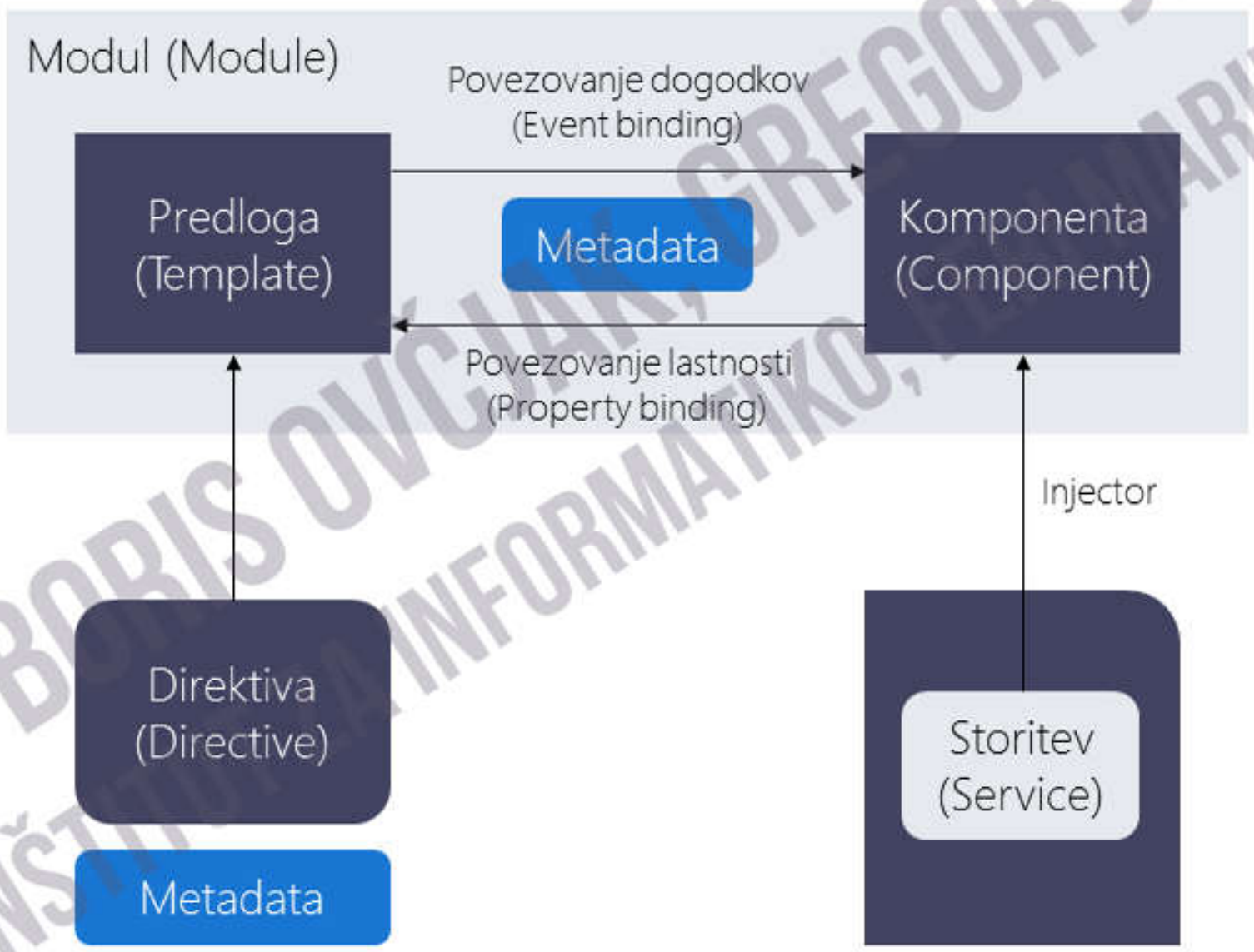
- V Angular se odvisnosti podajo kot argumenti komponentnega razrednega konstruktorja.
- Pri tem določimo tudi TypeScript tip, ki Angularju pove kaj vriniti tudi v primeru minifikacije.

ANGULARJS

ANGULAR

ARHITEKTURA ANGULAR APLIKACIJE

ARHITEKTURA ANGULAR APLIKACIJE



GLAVNI GRADNIKI ANGULAR

MODULI (MODULES)

KOMPONENTE (COMPONENTS)

DATA BINDING

PREDLOGE (TEMPLATES)

DIREKTIVE (DIRECTIVES)

METAPODATKI (METADATA)

SERVICES (STORITVE)

DEPENDENCY INJECTION

MODULI - MODULES

- Angular aplikacije so modularne,
- Moduli skrbijo za organizacijo aplikacije v kohezivni blok funkcionalnosti,
- Angular modul je razred z dekoraterjem `@NgModule`
  - Na osnovi meta-podatkovnega objekta pove ogrodju, kako prevesti in poganjati programsko kodo modula,
  - Identificira komponente, direktive in pipe ter jih postavi na voljo zunanjim komponentam.

- Vsaka aplikacija ima vsaj en modul:
  - En korenski modul – navadno `AppModule` (manjše aplikacije)
  - Več modulov funkcionalnosti (večje aplikacije)

MODULI - MODULES

PRIMER PREPROSTEGA ROOT MODULA

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from
'@angular/platform-browser';
@NgModule({
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  bootstrap:    [ AppComponent ]
})

export class AppModule { }
```

Razredi pogledov, ki pripadajo temu modulu. Angular vsebuje tri tipe razredov pogledov: komponente, direktive, pipe.

Podmnožica deklaracij, ki so vidne in uporabne predlogam komponent ostalih modulov.

Ostali moduli, katerih izvoženi razredi so potrebni s strani predlog komponente deklarirane v tem modulu.

Kreatorji storitev, ki jih ta modul poda na voljo globalni množici storitev – postanejo dostopni vsem delom aplikacije

Glavni aplikacijski pogled, kjer gostujejo vsi pogledi aplikacije. Le korenski modul mora nastavljati to lastnost.

KOMPONENTE- COMPONENTS

- Komponenta nadzoruje pogled ter vsebuje aplikacijsko logiko, vezano na pogled (znotraj razreda).
- Razred deluje nad pogledom preko API-jev lastnosti in metod.
- Angular ustvarja, posodablja in uničuje komponente glede na uporabniške premike po aplikaciji.

```
export class ElementListComponent implements OnInit {
  elementi: Element[];
  izbranEl: Element;

  constructor(private service: ElementService) { }
  ngOnInit() {  this.elementi = this.service.pridobiElemente(); }
  izbranEl(el: Element) { this.izbranEl = el; }
}
```

PRIMER KOMPONENTE

PREDLOGE - TEMPLATES

- Predloga je oblika HTML strani, ki pove Angular-ju kako naj izriše komponento.
- Predloga izgleda kot navadna HTML stran, ki pa vsebuje dodatne elemente Angular sintakse, npr.:
  - ***ngFor** – iteracija skozi polje elementov,
  - **{{object.lastnost}}** – izpis lastnosti,
  - **(click)** – dogodek klik,
  - **<podrobnosti>** - element po meri (predstavlja novo komponento), ...

```
<h2>{{naslov}}</h2>
<ul><li *ngFor="let el of elementi" (click)="izbranEl(el)">
   {{el.naziv}}
</li></ul>
<podrobnosti *ngIf="izbranEl" [element]="izbranEl"></podrobnosti>
```

PRIMER POGLEDA

METAPODATKI - METADATA

- Metapodatki povejo Angular-ju kako naj procesira razred.
  - Angular komponente so le razredi, dokler ne povemo Angular-ju da gre za komponento. Za to poskrbijo metapodatki.
  - Metapodatki se določijo z uporabo **dekoratorja.**

PRIMER METAPODATKOV - KOMPONENTA

```
@Component({
  moduleId: module.id,
  selector:    'seznam-elementov',
  templateUrl: './seznam-elementov.component.html',
  providers:  [ ElementService ]
})
export class ElementListComponent
implements OnInit { /* . . . */ }
```

Nastavi vir osnovnega naslova za relativen URL modula, kot je templateUrl

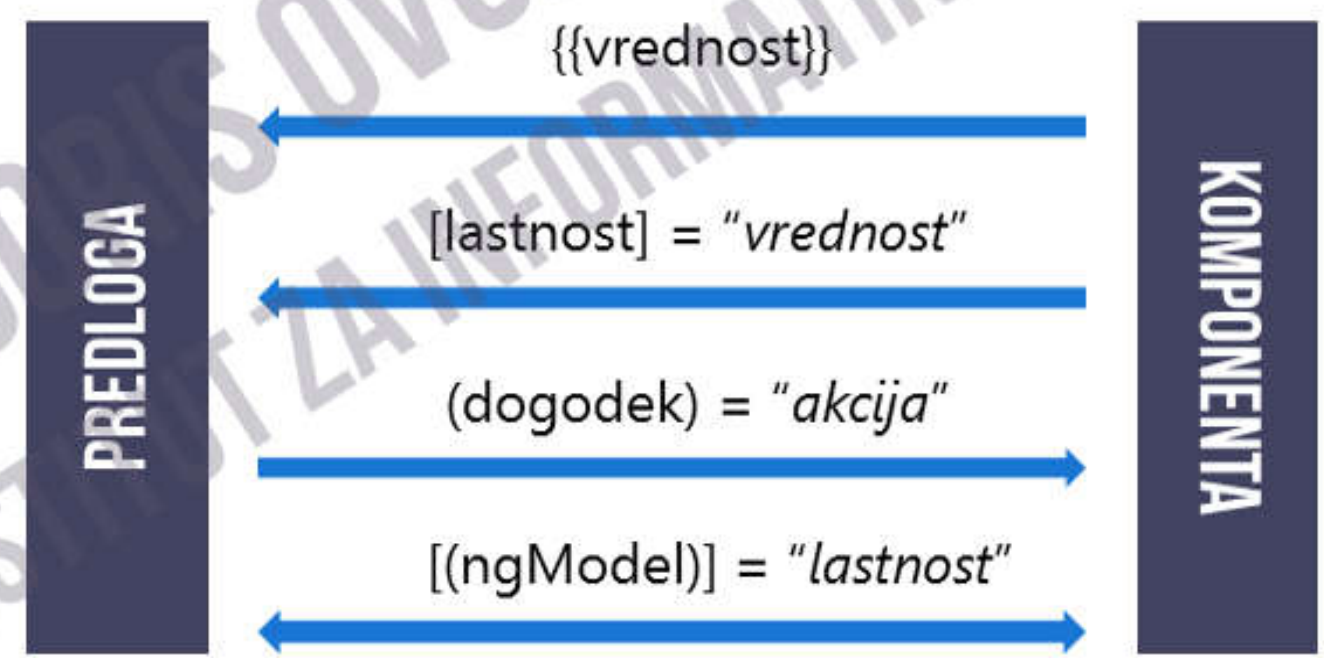
CSS selektor, ki pove Angular-ju da ustvari in vstavi instance te komponente, kjer najde to oznako

Relativen naslov komponentnega pogleda glede na modul

Polje ponudnikov storitev, ki jih potrebuje komponenta.

POVEZOVANJE PODATKOV – DATA BINDING

- Angular podpira **data binding**, mehanizem za povezovanje delov predloge z delom komponente.
- Obstajajo štiri oblike sintakse povezovanja podatkov. Vsaka ima obliko in usmeritev:



DIREKTIVE - DIRECTIVES

- Angular predloge so dinamične. Pri generiranju Angular oblikuje DOM glede na navodila s strani direktiv.
- Direktiva je razred z dekoratorjem `@Directive`.
  - Komponenta je direktiva s komponento (`@Component` dekorator je pravzaprav direktiva razširjena z lastnostmi, vezanimi na predlogo.

- Obstajata še dve vrsti direktiv:
  - Strukturne,
  - Atributne.

DIREKTIVE - DIRECTIVES

STRUKTURNE DIREKTIVE

Strukturne direktive spreminjajo postavitev z dodajanjem, odstranjevanjem in menjavo elementov v DOM.

```
<li *ngFor="let el of elementi"></li>

<podrobnosti *ngIf="izbranEl " [element]="izbranEl"></podrobnosti>
```

ATRIBUTNE DIREKTIVE

Atributne direktive spreminjajo izgled ali obnašanje obstoječega elementa. V predlog izgledajo kot obstoječi HTML atributi (ime).

```
<input [(ngModel)]="el.naziv">
```

STORITVE - SERVICES

- Storitve predstavljajo široko kategorijo vrednosti, funkcij ali funkcionalnosti, ki jih aplikacija potrebuje.
- Storitev je lahko karkoli, v praksi je navadno razred z ozkim, dobro definiranim specifičnim namenom, npr.:
  - Storitev beleženja (logiranja),
  - Podatkovne storitve,
  - Vodilo sporočanja,
  - Aplikacijske nastavitve, ...
- Angular nima posebne definicije za storitve (poseben nadrazred), prav tako storitev ni potrebno vnaprej registrirati.

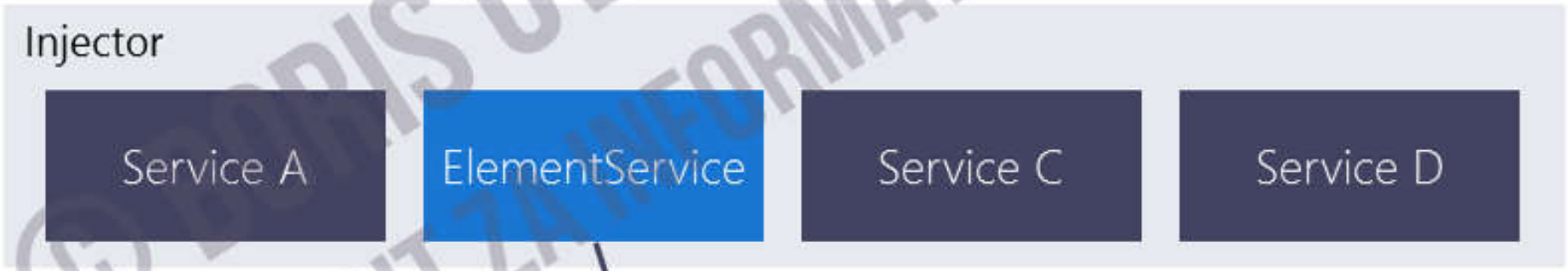
STORITVE - SERVICES

- Storitve koristijo predvsem komponente,
- Priporočeno je, da so komponente preproste – ne dostopajo neposredno do podatkov, validirajo vnosa ali logirajo neposredno v konzolo. Za takšne naloge so na voljo **storitve**.
- Naloga komponente je le, da omogoča uporabniško izkušnjo in nič več.
- **Angular ne zahteva upoštevanja teh principov!**

VRIVANJE ODVISNOSTI – DEPENDENCY INJECTION

- Vrivanje odvisnosti je način dobave nove instance razreda s polnimi odvisnostmi.
- Angular npr. uporablja vrivanje odvisnosti za dostavljanje novih komponent s storitvami, ki jih potrebujejo
  - Večina odvisnosti so storitve.

1 Injector vsebuje vsebnik storitvenih instanc, ki jih je že ustvaril. Če zahtevane storitve še ni v vsebniku, injector ustvari novo instance in jo doda v vsebnik preden vrne storitev Angularju.



3 Ko so vse zahtevane storitve razrešene, lahko Angular pokliče konstruktor s storitvami kot argumenti.

```
constructor(private service: ElementService) { }
```

2 Pri ustvarjanju komponente Angular najprej poišče **Injector** za storitev, ki jo komponenta potrebuje

VRIVANJE ODVISNOSTI – DEPENDENCY INJECTION

GLAVNE ZNAČILNOSTI:

- Vrivanje odvisnosti je ključen del ogrodja Angular in se uporablja povsod,
- Glavni mehanizem je Injector:
  - Upravlja vsebnik storitvenih instanc, ki jih je kreiral,
  - Lahko ustvari novo instance storitve, ki jih definira ponudnik.

```
export class Predmet {
  public naziv: string;
  public profesor: Profesor;
  public sestavaOcene: Ocena;

  constructor() {
    this.professor = new Profesor();
    this.sestavaOcene = new Ocena();
  }
}
```

```
export class Predmet {
  public naziv: string;


  constructor(public profesor: Profesor,
    public sestavaOcene: Ocena) {
      …
    }
}
```

RAZRED Z VRINJENO ODVISNOSTJO

PRIPRAVA, PREGLED IN ZAGON
ANGULAR APLIKACIJE

PRIPRAVA APLIKACIJE

1. Namestitev **Node.js** in **npm** (Node package manager)

2. Instalacija **Angular-CLI** (angl. Command Line Interface)

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

3. Izdelava novega projekta
   1. Odpremo novo terminalno okno
   2. Izdelamo nov aplikacijski projekt (Angular-CLI)

```
ng new ime-aplikacije  // Ukaz ustvari nov projekt (strukturo)
```

ANGULAR CLI

Cilj (bližnjice)	Ukaz
Projekt	`ng new project-name`
Komponenta	`ng g component my-new-component`
Direktiva	`ng g directive my-new-directive`
Pipe	`ng g pipe my-new-pipe`
Storitev	`ng g service my-new-service`
Razred	`ng g class my-new-class`
Vmesnik	`ng g interface my-new-interface`
Enum	`ng g enum my-new-enum`
Modul	`ng g module my-module`
Zagon strežnika	`ng serve`

PREGLED APLIKACIJE

- Aplikacija se nahaja v podmapi **src**

app
- app.component.css
- app.component.html
- app.component.spec.ts
- app.component.ts
- app.module.ts

Vsebuje komponento AppComponent, skupaj s HTML predlogo in stili (CSS).

Predstavlja korensko komponento, ki postane drevo gnezdenih komponent, ko se aplikacija razvija.

Definira AppModule, korenski model, ki pove Angularju, kako sestaviti aplikacijo

assets

Mapa, namenjena dodatnim elementom kot so slike ali karkoli, kar je potrebno prenesti ob zagonu aplikacije

environments

Mapa vsebuje datoteko za vsakega od končnih okolij, kjer vsak izvozi preproste konfiguracijske spremenljivke za aplikacijo

PREGLED APLIKACIJE

index.html — Glavna vstopna stran, ki jo predstavi strežnik. V večini primerov je ni potrebno urejati. CLI samodejno doda .js in .css datoteke, ko prevedete aplikacijo. Tako ni potrebno nikoli ročno dodajati značk <script> ali <link>.

main.ts — Glavna vstopna točka aplikacije. Prevede aplikacijo s pomočjo JIT ali AoT prevajalnika in postavi (angl. Bootstraps) aplikacijski korenski modul v brskalnik.

polyfills.ts — Datoteka, namenjena uravnavanju različnih podpor spletnim standardom s strani različnih brskalnikov.

styles.css — Globalni stili za aplikacijo

test.ts — Glavna vstopna točka za unit teste.

tsconfing.json — Konfiguracija TypeScript prevajalnika.

PREGLED APLIKACIJE

- Korenski direktorij aplikacije – mapa "**/ime-aplikacije**"

e2e/*	Vsebuje End-to-end teste. Nahajajo se zunaj mape src, saj gre za ločeno aplikacijo, ki testira glavno aplikacijo.
node_modules/...	Mapa, krenirana s strani Node.js. V njej se nahajajo moduli, predstavljeni v datoteki package.json.
angular-cli.json	Konfiguracija Angular-CLI.
package.json	Seznam npm konfiguracij tretjeosebnih paketov, ki jih uporablja projekt.

ZAGON APLIKACIJE

- Zagon aplikacije z uporabo Angular-CLI

```
cd ime-aplikacije // Navigacija v mapo s projektom
ng serve // Zagon strežnika (lite-server)
```

- Ukaz **ng serve** skrbi za postavitev aplikacije na testni strežnik (lite-server):
  - Zažene strežnik,
  - Opazuje datoteke,
  - Ob spremembah datotek avtomatično prevede aplikacijo in osveži brskalnik.

KORENSKI MODUL

ROOT MODULE

KORENSKI MODUL- ROOT MODULE

- Vsaka Angular aplikacija mora imeti korenski modul.
- Korenski modul je razred, ki opredeli, kako je aplikacija sestavljena.
- Konvencionalno se imenuje **AppModule**.

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

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

KORENSKI MODUL- @NGMODULE

- Dekorator @NgModule označi **AppModule** kot Angular modularni razred (NgModule class).
- Dektorator prejme meta-podatkovni objekt, ki opredeli, kako Angular pretvori izvorno kodo (compile) in zažene aplikacijo.
- Metadata objekt je sestavljen iz naslednjih lastnosti:
  - **imports** – specifični Angular moduli, ki jih potrebujemo za izvajanje.
  - **declarations** – opredelimo vse komponente, ki se izvajajo v aplikaciji.
  - **bootstrap** – korenska komponenta, ki jo Angular ustvari in vključi v **index.html**.
  - *exports* – je namenjen definiranju komponent, ki jih modul izvozi, da jih lahko potem uvozimo v drugih modulih. Korenski modul načeloma ne potrebuje *exports*.

KORENSKI MODUL- IMPORTS

- Polje `import` vsebuje nabor Angular modulov (razredi `@NgModule`), ki jih aplikacija potrebuje za delovanje.
- Večina funkcionalnosti Angular je opredeljena v obliki modulov, npr. HTTP storitve so v `HttpModule`.
- Če delamo spletno aplikacijo, moramo imeti v polju `import` vedno `BrowserModule` (vključuje običajne direktive, kot npr. ngIf in ngFor).

KORENSKI MODUL- DECLARATIONS

- Vsaka komponenta mora biti deklarirana v razredu `NgModule`.
  - Če želimo komponento uporabljati, brez da jo prej deklariramo, Angular aplikacija proži napako.
- To dosežemo s pomočjo polja `declarations`.
- Polje `declarations` lahko vsebuje komponente, direktive in *pipes*.
- Ne sme vsebovati razredov `@NgModule`, storitev (te so opredeljene pod *providers*) ali modelov.

KORENSKI MODUL- BOOTSTRAP

- Angular aplikacija se zažene tako, da se naloži korenski `AppModule`.
- Ustvarijo se komponente, opredeljene v polju `bootstrap` in se nato vstavijo v DOM.
- Najbolj običajno imamo samo eno korensko komponento, ki se konvencionalno imenuje `AppComponent`.

KORENSKI MODUL- DINAMIČNI ZAGON

- Angular aplikacija se najprej dinamično prevede s pomočjo *Justin-in-Time* (JIT) prevajalnika in se požene v brskalniku.
- Običajno se zagon aplikacije izvede v ločeni datoteki z imenom **`main.ts`**, znotraj imenika **`src`**:

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

KORENSKI MODUL- DINAMIČNI ZAGON

- Zagon poteka v naslednjih korakih:
  1) koda v **`src/main.ts`** vzpostavi okolje za izvajanje,
  2) nato vzame **`AppComponent`** iz polja **`bootstrap`**,
  3) ustvari instanco komponente in
  4) jo vstavi v HTML značko, opredeljeno v **`selector`** delu komponente.
- Običajno je takšen **`selector`** opredeljen kot **`<my-app>`**.

TYPESCRIPT

TS TYPESCRIPT

- Microsoftov brezplačen in odprtokoden meta JavaScript jezik
  - Se prevede v JavaScript kodo
  - Poenostavitev sintakse. razširitev funkcionalnosti, povečana zmogljivost
- Najnovejša različica: 2.3 (27. april 2017).
- Ima podobno sintakso JavaScript-u
  - Omogoča mešanje TypeScript in JavaScript kode
  - Kompatibilen z ECMAScript6 sintakso
- Dodatne lastnosti
  - Statično strogo tipiziranje
  - Razredi, vmesniki, moduli, generiki
  - Izvrstna orodja za razvoj (Visual Studio)
  - *Angular ga uporablja kot primarni jezik*

TS TYPESCRIPT - UPORABA

- TypeScript datoteke imajo praviloma .ts končnico
- Izvorno kodo je potrebno prevesti v JavaScript kodo
  - Prevajalnik je na voljo v obliki Node.js paketa
  - Namestitev: **`npm install -g typescript`**
  - Uporaba: **`tsc helloworld.ts`**
  - Vzpostavitev v VS Code.
- Obstaja tudi uradno spletno razvojno okolje
  - http://www.typescriptlang.org/Playground

TS TYPESCRIPT - TIPIZIRANJE

JavaScript kompatibilni podatkovni tipi

- Boolean (`let isDone: boolean= false;`)
- Number (`let decimal: number= 6;`)
- String (`let fullName: string= `Bob Bobbington`;`)
- array (dva načina uporabe - Tip[] ali Array<Tip>)
- any (karkoli, s tem se izognemo strogemu tipiziranju: *"opt-in and opt-out of type-checking during compilation"*)

Dodatni podatkovni tipi

- void (brez vrednosti, uporabno le pri metodah)
- never (za funkcije, ki vrnejo napako)
- enum
  - Uporaba: `enum Spol {M, Ž};`

TS TYPESCRIPT – UPORABA TIPIZIRANJA

JAVASCRIPT

```
function Sestej(stevilo1, stevilo2) {
      return stevilo1 + stevilo2;
}

let rezultat1 = Sestej(5, 3);
console.log(rezultat1);
// 8

let rezultat2 = Sestej('Janez', 'Novak');
console.log(rezultat2);
//JanezNovak
```

TYPESCRIPT

```
function Sestej(stevilo1: number, stevilo2:
number): number {
    return stevilo1 + stevilo2;
}

let rezultat1: number = Sestej(5, 3);
console.log(rezultat1);
// 8

let rezultat2: number = Sestej('Janez',
'Novak');
console.log(rezultat2);
//primer.ts(9,32): error TS2345:
//Argument of type '"Janez"' is not
assignable to parameter of type 'number'.
```

TS TYPESCRIPT – VMESNIKI

Definiranje lastnosti

```
[ts]
Type '{ naslov: string; kljucneBesede: string[]; }' is not assi
gnable to type 'IDelavnica'.
  Property 'id' is missing in type '{ naslov: string; kljucneBe
sede: string[]; }'.
```

```
interface IDelavnica {
  id: number;
  naslov: string;
  vsebina?: string;
  kljucneBesede: string[]
}

const izpis = (delavnica: IDelavnica) => {
  console.log(`Delavnica z ID-jem ${delavnica.id},
              naslovom ${delavnica.naslov}
              in ključnimi besedami ${delavnica.kljucneBesede.join(', ')}`);
};

let podatek: IDelavnica = {
  id: 1,  ←—  Brišemo
  naslov: 'Uvod v Angular',
  kljucneBesede: ['JavaScript', 'Angular', 'TypeScript']
};

izpis(podatek);
```

TS TYPESCRIPT – VMESNIKI

Implementacija vmesnika

```
interface OnInit {
  ngOnInit(): void;
}

class FilmKomponenta implements OnInit {

  constructor() {
    this.ngOnInit();
  }

  ngOnInit() {
    console.log('Pozdravljen, svet!')
  }
}

let film1 = new FilmKomponenta();
```

TS TYPESCRIPT – RAZREDI

- Sintaksa definicije razredov je kompatibilna z ECMAScript6
- Podobno kot v ostalih programskih jezikih, lahko tudi tukaj dedujemo in implementiramo vmesnike
- Vidljivost spremenljivk (private, public)
- Podpora statičnim lastnostim in funkcijam
- Podpira abstraktne razrede

```
class Film {
  private zvrsti: string[];
  constructor(public naziv: string, zvrstiFilma: string[]) {
    this.zvrsti = zvrstiFilma;
  }

  get Zvrsti() {
    return this.zvrsti.join(', ');
  }
}

let ww = new Film('Wonder Woman', ['Action', 'Adventure', 'Fantasy']);

console.log(ww.naziv, ww.Zvrsti);
//Wonder Woman Action, Adventure, Fantasy
```

TS TYPESCRIPT – RAZREDI

NADRAZRED

```
class Oseba {
  constructor(
    public ime: string,
    public spol: string) { }

  Podrobnosti(): string {
    return `Oseba ${this.ime} je
            ${this.spol}`;
  }
}
```

PODRAZRED

```
class Igralec extends Oseba {
  constructor(
    ime: string,
    spol: string,
    public filmi: string[]) {
      super(ime, spol);
    }

  Podrobnosti(): string {
    return `Seznam filmov ${this.ime}:
            ${this.filmi.join(', ')}`;
  }
}
```

UPORABA

```
let janezNovak = new Oseba('Janez Novak', 'moški'),
galGadot = new Igralec('Gal Gadot', 'ženska', ['Wonder Woman', 'Triple 9']);
console.log(janezNovak.Podrobnosti()); //Oseba Janez Novak je moški
console.log(galGadot.Podrobnosti()); //Seznam filmov Gal Gadot: Wonder Woman, Triple 9
```

TS TYPESCRIPT – RAZREDI

ES6

NADRAZRED

```
class Oseba {
  constructor(ime, spol) {
    this.ime = ime;
    this.spol = spol;
  }
  Podrobnosti() {
    return `Oseba ${this.ime} je
            ${this.spol}`;
  }
}
```

PODRAZRED

```
class Igralec extends Oseba {
  constructor(ime, spol, filmi) {
    super(ime, spol);
    this.filmi = filmi;
  }

  Podrobnosti() {
    return `Seznam filmov ${this.ime}:
            ${this.filmi.join(', ')}`;
  }
}
```

UPORABA

```
let janezNovak = new Oseba('Janez Novak', 'moški'),
galGadot = new Igralec('Gal Gadot', 'ženska', ['Wonder Woman', 'Triple 9']);
console.log(janezNovak.Podrobnosti()); //Oseba Janez Novak je moški
console.log(galGadot.Podrobnosti()); //Seznam filmov Gal Gadot: Wonder Woman, Triple 9
```

TS TYPESCRIPT – RAZREDI

ES5 classic

```
var __extends = (this && this.__extends) || (function () {
    var extendStatics = Object.setPrototypeOf ||
        ({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ = b; }) ||
        function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; };
    return function (d, b) {
        extendStatics(d, b);
        function __() { this.constructor = d; }
        d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype, new __());
    };
})();
var Oseba = (function () {
    function Oseba(ime, spol) {
        this.ime = ime;
        this.spol = spol;
    }
    Oseba.prototype.Podrobnosti = function () {
        return "Oseba " + this.ime + " je " + this.spol;
    };
    return Oseba;
}());
var Igralec = (function (_super) {
    __extends(Igralec, _super);
    function Igralec(ime, spol, filmi) {
        var _this = _super.call(this, ime, spol) || this;
        _this.filmi = filmi;
        return _this;
    }
    Igralec.prototype.Podrobnosti = function () {
        return "Seznam filmov " + this.ime + ": " + this.filmi.join(', ');
    };
    return Igralec;
}(Oseba));
var janezNovak = new Oseba('Janez Novak', 'moški'), galGadot = new Igralec('Gal Gadot', 'ženska', ['Wonder Woman', 'Triple
9']);
console.log(janezNovak.Podrobnosti()); //Oseba Janez Novak je moški
console.log(galGadot.Podrobnosti()); //Seznam filmov Gal Gadot: Wonder Woman, Triple 9
```

TS TYPESCRIPT – MODULI

- Namen
  - Imajo vlogo imenskih prostorov– izvajajo se samo znotraj svojega obsega
  - Preprečevanje imenskih konfliktov
  - Ob pravilni uporabi dosežemo večjo preglednost in modularnost
- Vsebina modula ni privzeto vidna navzven
  - Za to imamo na voljo ukaz export
  - Modul uporabimo z ukazom import
- Izvozimo lahko spremenljivko, razred, funkcijo, vmesnik...
- Vsaka datoteka, ki vsebuje export oz. import, predstavlja modul.
- Dve vrsti modulov
  - **Interni** - privzeto, ustreza večini primerov uporabe
  - **Eksterni** - ob uporabi z Node.JS ali Require.js
  - *It's important to note that in TypeScript 1.5, the nomenclature has changed. "**Internal modules**" are now "**namespaces**". "**External modules**" are now simply "**modules**", as to align with ECMAScript 2015's terminology*

TS TYPESCRIPT – MODULI

INTERFACE.TS

```ts
export interface IOseba {
  ime: string;
  priimek: string;
  Izpis(): string;
}
```

RAZRED.TS

```ts
import { IOseba } from "./interface";
export class Igralec implements IOseba {
  constructor(public ime: string, public priimek: string) { }

  public Izpis(): string {
    return `Sem ${this.ime} ${this.priimek}`;
  }
}
```

MAIN.TS

```ts
import { Igralec as Actor } from "./razred";
let galGadot = new Actor('Gal', 'Gadot');
console.log(galGadot.Izpis()); //Sem Gal Gadot
```

TS TYPESCRIPT – FUNKCIJE

- Funkcije so najpomembnejši gradnik JavaScripta
- Funkcijo lahko definiramo kot
  - Funkcijo
  - Spremenljivko

```
function Funkcija() { }
const es5SpremenljivkaKotFunkcija = function () { };
const fatArrowFunkcija = () => { };
const namedFunctionExpression = function moja() { };
```

- Dodatne funkcionalnosti v TypeScript-u
  - Strogo tipiziranje
  - Definiranje funkcijskega tipa (podobno vmesniku)
  - Omogoča opcijske parametre
  - Omogoča privzeto vrednost parametrov

TS TYPESCRIPT – FUNKCIJE

PRIMER 1: TIPIZIRANJE

```
function sestej(a: number, b: number): number {
  return a + b;
}
```

PRIMER 2: FUNKCIJSKI TIP

```
// Implicitni vmesnik funkcije
let _sestej: (a: number, b: number) => number;
_sestej = function (x: number, y: number): number {
  return x + y;
};
```

TS TYPESCRIPT – FUNKCIJE

- Opcijski in privzeti parametri
  - Za ime opcijskega parametra napišemo vprašaj ?
  - Privzeto vrednost napišemo kot prirejanje v glavi funkcije
  - Parametra ne moremo hkrati označiti kot opcijskega in mu določiti privzeto vrednost

PRIMER 1: OPCIJSKI PARAMETER

```
function oseba(ime: string, priimek?: string) {
  return `${this.ime} ${this.priimek}`;
}
let result1 = oseba("Gal");
let result2 = oseba("Gal", "Gadot");
```

PRIMER 2: PRIVZETI PARAMETRI

```
function pozdrav(sporocilo: string = "Hello, World!") { }
```

DELO S PODATKI

DELO S PODATKI- INTERPOLACIJA

- V osnovi se podatki Angular aplikacije prikazujejo tako, da lastnost komponente izpišemo v vnaprej določeni obliki znotraj predloge HTML.
- Slednje najlažje dosežemo z interpolacijo („vrinek“) – lastnost komponente se znotraj predloge HTML ovije v dvojne zavite oklepaje, npr.: `{{ime}}`
- Primer.
  - Angular samodejno pobere lastnosti iz komponente in jih vstavi v „brskalnik“ ter posodobi prikaz, ko se te lastnosti spremenijo.
  - Nikjer ne ustvarimo novo instanco komponente, ker za to poskrbi Angular samodejno (v času zagona – bootstrap).

DELO S PODATKI- IZRAZ PREDLOGE

- V splošnem se vsebina med zavitimi oklepaji imenuje „izraz predloge“ (ang. template expression), ki se najprej preveri in nato pretvori v niz.

```
<!-- "Seštevek 1 + 1 je 2" -->
<p>Seštevek 1 + 1 je {{1 + 1}}</p>
```

- Izraz lahko tudi pokliče metode pripadajoče komponente.
- Angular izvede izraz in priredi vrednost lastnosti, ki pa je lahko element HTML, komponenta ali direktiva.
- Izrazi se pišejo v jeziku, podobnem jeziku JavaScript in prepovedujejo uporabo:
  - prirejanja (=, +=, -=, ...)
  - new
  - Veriženja izrazov z ; ali ,
  - povečevanja ali zmanjševanja (++ in --)

DELO S PODATKI- IZRAZ PREDLOGE

- Kontekst izraza je običajno instanca komponente (ki je izvor vrednosti).
  - V primeru `{{ime}}` predstavlja ime lastnosti komponente.
  - V primeru `[disabled]="onemogocen"` se sklicujemo na komponento oz. njeno lastnost `onemogocen`.
  - Primer.
- Izrazi se ne morajo sklicevati na globalni imenski prostor (`window`, `document`, `console.log`, `Math.max` itd.).

DELO S PODATKI- IZRAZ PREDLOGE

- **Referenčna spremenljivka predloge** predstavlja referenco na element DOM znotraj predloge (lahko je tudi referenca na Angular komponento ali direktivo).
- Na referenčo spremenljivko predloge se lahko sklicujemo kjerkoli znotraj predloge.
- Takšno spremenljivko označimo z znakom **#** in se lahko definira zgolj enkrat znotraj celotne predloge.
- Primer.

DELO S PODATKI- IZRAZ PREDLOGE

- Izrazi predloge imajo na voljo tudi operator **`pipe`**, ki omogoča manjše transformacije vrednosti.
- **`Pipe`** so preproste funkcije, ki prejmejo vhodno vrednost in vrnejo preoblikovano vrednost.
  - `<div>Naslov preko pipe: {{naslov | uppercase}}</div>`
- Na voljo imamo običajne (manjše) transformacije:
  - `uppercase`
  - `lowercase`
  - `json`
  - `titlecase`

DELO S PODATKI- ODRAZ PREDLOGE

- Za razliko od izraza predloge se odraz predloge (angl. template statement) odzove na dogodek, ki ga proži element, komponenta ali direktiva.

```
<button (click)="Funkcija()">Woops!</button>
```

- Namen odraza predloge je, da ima *„stranski učinek“*, tj. posodobitev stanja aplikacije.
- Podobno kot pri izrazu predloge tudi v odrazu predloge uporabljamo jezik, podoben jeziku javaScript.
  - Razlika: lahko uporabljamo preproste prireditve (=) in veriženje izrazov.

DELO S PODATKI- VEZANJE PODATKOV

- Vezanje podatkov (ang. data binding) pomeni, da ogrodje (Angular) samo skrbi za branje in pisanje podatkov v sklopu dokumenta HTML.
  - Naša naloga je, da zagotovimo vezavo med izvorom in ciljem.
- Poznamo več vrst vezanja podatkov, v grobem pa jih lahko razvrstimo v tri skupine, glede na smer prenosa podatkov:
  - Enosmerno od izvora do pogleda (One-way from data source to view target)
  - Enosmerno od pogleda do izvora (One-way from view target to data source)
  - Dvosmerno (Two-way)

DELO S PODATKI- VEZANJE PODATKOV

Smer podatkov	Sintaksa (Angular)	Tip vezave

Enosmerno od podatkovnega vira do cilja pogleda	`{{expression}}`
	`[target] = "expression"`
	`bind-target = "expression"`
		Interpolation
		Property
		Attribute
		Class
		Style

Enosmerno od pogleda cilja do podatkovnega vira	`(target) = "statement"` `on-target = "statement"`	Event
Dvosmerno	`[(target)] = "expression"` `bindon-target = "expression"`	Two-way

Razen interpolacije vsi ostali tipi vezave ovijejo cilj (target) v oklepaje [ ], ( ) ali pa se cilj prične s predpono (bind-, on-, bindon-).

Cilj je v vsakem primeru *lastnost* in *ne atribut* elementa.

DELO S PODATKI- VEZANJE PODATKOV

- Vezanje podatkov na lastnost ali atribut dokumenta HTML?
  - `<button [disabled]="nespremenjen">Shrani</button>`
- Ali je interpretacija zgornjega primera pravilna? „Vežemo se na **`disabled` atribut** elementa HTML in mu priredimo vrednost spremenljivke **`nespremenjen`** (**`true`** / **`false`**)“
- Ko vežemo podatke na elemente HTML, ne naslavljamo več atributov, ampak lastnosti DOM elementa!

DELO S PODATKI- VEZANJE PODATKOV

- Atribut HTML ali lastnost DOM?
  - Atribute definira HTML, lastnosti definira DOM (Document Object Model).
- Atributi inicializirajo lastnosti DOM - vrednosti lastnosti DOM se lahko spreminjajo, vrednosti atributov ne.
  - Atribut torej predstavlja začetno vrednost, lastnost pa trenutno.
  - Nekaj atributov HTML ima neposredno preslikavo v lastnosti (npr. **`id`**).
  - Nekateri atributi HTML nimajo pripadajočih lastnosti (npr. **`colspan`**).
  - Nekatere lastnosti DOM nimajo pripadajočih atributov (npr. **`textContent`**).

DELO S PODATKI- VEZANJE PODATKOV

- Atribut HTML in lastnost DOM nista ista stvar, čeprav lahko imata isto ime. Primer.
- Ko pišemo kodo HTML lahko definiramo *atribute* na elementih HTML in ko brskalnik razčleni (angl. parse) kodo HTML, se ustvari pripadajoči DOM.
    - Ker je DOM objekt, ima *lastnosti*.
- Vezanje podatkov v Angular deluje nad lastnostmi DOM in dogodki, ne nad atributi! Ali pač?

DELO S PODATKI- CILJI VEZANJA

- Cilj vezanja podatkov je vedno nekaj, kar je del DOM.
- Odvisno od tipa vezanja je cilj lahko:
  - lastnost (element, komponenta, direktiva),
  - dogodek (element, komponenta, direktiva) ali
  - (redko) atribut.

DELO S PODATKI- CILJI VEZANJA

```
<img [src]="heroImageUrl">
```
LASTNOST ELEMENTA

```
<hero-detail [hero]="currentHero"></hero-detail>
```
LASTNOST KOMPONENTE

```
<div [ngClass]="{special: isSpecial}"></div>
```
LASTNOST DIREKTIVE

```
<button (click)="shrani()">Save</button>
```
DOGODEK ELEMENTA

```
<fitness (deleteRequest)="izbrisi()"></fitness>
```
DOGODEK KOMPONENTE

```
<div (mojClick)="clicked=$event" clickable>klik</div>
```
DOGODEK DIREKTIVE

```
<input [(ngModel)]="ime">
```
DVOSMERNO POVEZOVANJE

```
<button [attr.aria-label]="help">help</button>
```
LASTNOST ATRIBUTA (IZJEMA)

```
<div [class.special]="isSpecial">Special</div>
```
LASTNOST CLASS

```
<button [style.color]="isSpecial ? 'red' : 'green'">
```
LASTNOST STILA CSS

DELO S PODATKI- VEZANJE LASTNOSTI

- Vezanje lastnosti se smatra kot enostransko vezanje podatkov; potekajo v eno smer, in sicer iz lastnosti direktive v lastnost ciljnega elementa.
- Podatke lahko v tem primeru samo nastavimo, ne moremo jih brati.
- Prav tako ni možno z vezanjem lastnosti klicati metodo ciljnega elementa (v ta namen imamo na voljo vezanje dogodkov).
- Ciljna lastnost se zapiše v oglatih oklepajih `[ ]` ali s predpono **`bind-`**
- Cilj je vedno lastnost DOM, četudi zgleda kot atribut HTML.
  - `<img [src]="urlSlike">`
- Pri nastavljanju lastnosti ne smemo prirejati vrednosti, jih povečevati itd.

DELO S PODATKI- VEZANJE LASTNOSTI

- Oglati oklepaji pri vezanju lastnosti skrbijo, da se vsebina preveri.
- Brez oglatih oklepajev se vsebina pretvori v navaden niz (string), kar lahko povzroči napako pri izvajanju.

```
<!-- ERROR: HeroDetailComponent.hero expects a
Hero object, not the string "currentHero" -->
<hero-detail hero="currentHero"></hero-detail>
```

- Oglate oklepaje lahko izpustimo, če želimo, da je vrednost lastnosti string ali pa je vrednost konstanta.

DELO S PODATKI- VEZANJE LASTNOSTI

- Vezanje lastnosti ali interpolacija?
- Pri prikazovanju podatkov ni razlike!

```
<img src="{{slikaUrl}}">
<img [src]="slikaUrl">
<span>{{naslov}}</span>
<span [innerHTML]="naslov"></span>
```

- Tudi iz vidika varnosti delujeta identično (ne dopuščata, da zlonamerna koda pride do brskalnika).

DELO S PODATKI- VEZANJE ATRIBUTOV

- Kot že rečeno, je vezanje atributov izjema, ki deluje samo v primeru, da atribut HTML nima „soimenjaka“ v lastnosti DOM.
  - Element TD nima colspan *lastnosti*, ampak colspan *atribut*.
  - Primer.

```
<tr>
  <td colspan="{{zdruzi}}">Sum: $180</td>
</tr>
<!--
  Can't bind to 'colspan'
  since it isn't a known property of 'td'
-->
```

  - Rešitev: `<td [attr.colspan]="zdruzi">Sum: $180</td>`

DELO S PODATKI- VEZANJE RAZREDOV

- Podobno kot vezanje atributov tudi vezanje razredov zahteva, da znotraj oglatih oklepajev uporabimo rezervirano besedo class (nadomesti obstoječe stile).

```
<div class="alert alert-info" [class]="'alert alert-
success'">This alert needs your attention</div>
```

- Če za besedo class napišemo še ime dejanskega razreda, lahko postavljamo pogoje, kdaj se razred upošteva.

```
<div [class.alert-success]="jeUspesno">This alert needs
your attention</div>
```

- Za dodajanje in odstranjevanje več razredov je bolj pravilno uporabiti ngClass.

DELO S PODATKI- VEZANJE STILOV

- Tudi stile lahko vežemo na elemente z uporabo oglatih oklepajev in rezervirane besede style.

```
<span [style.font-weight]="700">Wonder Woman</span> received
largely positive reviews.
```

- Lahko opredelimo tudi pogoje.

```
Wonder Woman received largely <span [style.color]="jePozitivno ?
'green' : 'red'">positive</span> reviews.
```

- Za dodajanje in odstranjevanje več stilov je bolj pravilno uporabiti ngStyle.

DELO S PODATKI- VEZANJE DOGODKOV

- Dogodki omogočajo tok podatkov iz elementa v komponento.
  - Vezanje lastnosti je potekalo iz komponente na element.
- Uporabniki lahko tipkajo, klikajo na miško itd.; s pomočjo vezanja dogodkov lahko upravljamo s temi dogodki.
- Sintaksa vezanja dogodkov je sestavljena iz ciljnega dogodka in izrazom predloge.

```
<button (click)="shrani()">Shrani</button>
```

Ciljni dogodek

Izraz predloge

DELO S PODATKI- VEZANJE DOGODKOV

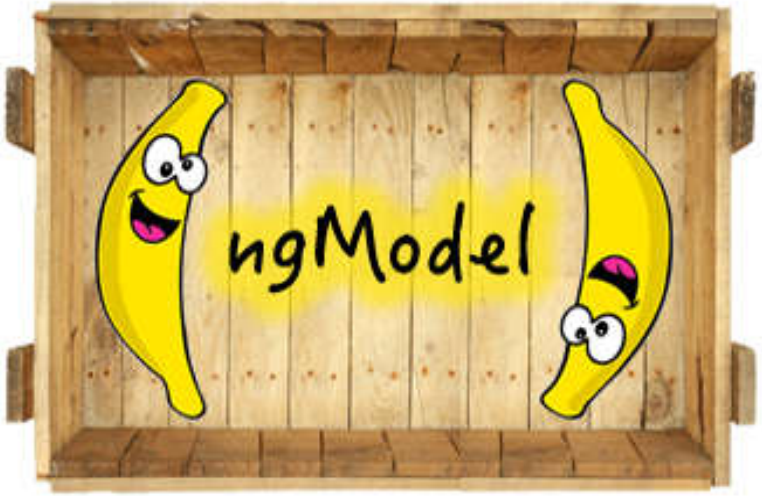
- Pri vezanju dogodkov Angular poskrbi za upravljalca dogodkov za ciljni dogodek.
- Ko se pojavi dogodek, upravljalec izvede pripadajoč izraz predloge.
- Preko vezanja se prenesejo informacije o dogodku, vključno s podatkovnimi vrednostmi, v obliki dogodkovnega objekta, imenovanega **`$event`**.
  - Če gre za DOM dogodek, je **`$event`** običajni DOM objekt dogodka, z lastnostma, kot sta **`target`** in **`target.value`**.
  - Če gre za dogodek direktive, je **`$event`** v obliki, kot jo opredelimo sami.

DELO S PODATKI- VEZANJE DOGODKOV

- Direktive preko **`EventEmitter`** prožijo dogodke po meri (custom events):
    1) Direktiva ustvari **`EventEmitter`** in ga izpostavi kot lastnost.
    2) Direktiva kliče **`EventEmitter.emit(payload)`**, da proži dogodek in pošlje vsebino (angl. payload), ki je lahko karkoli.
    3) Starševske direktive poslušajo za dogodek tako, da se vežejo na to lastnost in dostopajo do vsebine preko **`$event`** objekta.
- Primer dogodka po meri.

DELO S PODATKI- DVOSMERNO VEZANJE

- Dvosmerno vezanje podatkov pomeni, da prikazujemo vrednost lastnosti, ki se posodobi, ko uporabnik zada spremembe.
- Gre za kombinacijo nastavljanja specifične lastnosti elementa in poslušanja na spremembe (dogodek).
- Angular v ta namen ponuja posebno sintakso `[(x)]`
  - Vezanje lastnosti `[x]`
  - Vezanje dogodkov `(x)`



DELO S PODATKI- DVOSMERNO VEZANJE

- Sintaksa `[(x)]` pomeni, da imamo lastnost `x` in pripadajoč dogodek, poimenovan `xChange`.
- Primer
  - stevec → lastnost
  - stevecChange → dogodek
- Takšen pristop ni primeren za elemente `form` (`<input>` in `<select>`), v ta namen se uporablja `NgModel`.

DELO S PODATKI- INPUT, OUTPUT

- Iz prejšnjih primerov je razvidno, da smo uporabili dekoratorja `input` in `output`.
- Oba dekoratorja se uporabljata v primeru, ko želimo podati cilj vezanja pod-komponente (lastnosti elementa).
- `Input` lastnosti prejmejo vrednosti, opredeljene v predlogi.
- `Output` izpostavi EventEmitter objekte, dogodki tečejo (stream) iz lastnosti in jih prestreže upravljalec dogodkov, opredeljen v predlogi.

```
<moj-stevec [stevec]="vrednost" (stevecChange)="vrednost=$event"></moj-stevec>
```

Input

Output

VGRAJENE DIREKTIVE

<directive
...

VGRAJENE DIREKTIVE- UVOD

- Angular 1.x ima vgrajenih ~70 direktiv, skupnost pa jih je prispevala še več.
- Zaradi dobre strukture Angular ogrodja teh direktiv ni več toliko.
- V sklopu vgrajenih direktiv bomo predstavili najpogosteje uporabljene direktive, razdeljene v dve skupini:
  - Atributne direktive
  - Strukturne direktive

<directive
...

VGRAJENE DIREKTIVE – ATRIBUTNE DIREKTIVE

- Atributne direktive poslušajo in spreminjajo obnašanje elementov HTML, atributov, lastnosti in komponent.
- Elementom HTML jih pripnemo podobno kot atribute, zatorej tudi takšno ime.
- Najpogosteje uporabljene atributne direktive :
  1) `NgClass` dodaja in odstrani razrede CSS.
  2) `NgStyle` dodaja in odstrani stile HTML.
  3) `NgModel` dvosmerno vezanje podatkov na elementih forme HTML.

<directive
…

VGRAJENE DIREKTIVE – ATRIBUTNE DIREKTIVE

- Direktiva **ngModel** se uporablja za elemente forme HTML.
- Namen je, da prikazujemo podatke in ob spremembi te podatke posodobimo.
- Pred uporabo **ngModel** moramo uvoziti (**import**) **FormModule** in ga dodati v polje **imports** Angular modula.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'; // <--- JavaScript import iz Angular

@NgModule({
imports: [BrowserModule, FormsModule // <--- import v NgModule
],
/* ostali meta-podatki */
}) export class AppModule { }
```

<directive
...

VGRAJENE DIREKTIVE – ATRIBUTNE DIREKTIVE

- Lastnost **ngModel** nastavi vrednost elementu, medtem ko se dogodek **ngModelChange** proži ob vsaki spremembi vrednosti.
- Običajno ju zdržimo skupaj v **[(ngModel)]**, kar olajša vzpostavitev dvosmernega vezanja podatkov.
- Primer različnih pristopov.

<directive
...

VGRAJENE DIREKTIVE – STRUKTURNE DIREKTIVE

- Strukturne direktive so zadolžene za postavitev strani HTML.
- Z dodajanjem, odstranjevanjem in manipulacijo elementov, na katere so pripete, oblikujejo oz. preoblikujejo strukturo DOM.
- Pri uporabi strukturnih direktiv običajno uporabljamo znak * pred imenom, npr. `*ngIf`.
- Na posamezni element lahko dodamo največ eno strukturno direktivo.
- Najpogosteje uporabljeni strukturni direktivi sta `ngIf` in `ngFor`.

<directive
...

VGRAJENE DIREKTIVE- STRUKTURNE DIREKTIVE

- Direktiva **ngIf** omogoča dodajanje oz. odstranjevanje elementov iz DOM.

  - Primer: `<div *ngIf="aktiven; else loading "></div>`
  - Ko ima **aktiven** vrednost **truthy**, se doda DIV element, ko je **falsy** pa se odstrani.
- Ni isto kot prikazovanje in skrivanje; v primeru **ngIf** se elementi (in pod-elementi, komponente itd.) odstranijo iz DOM
  - V primeru skrivanja ostanejo elementi v DOM, ampak niso vidni .
  - **ngIf** izboljša performanco (zadeve se „fizično" odstranijo iz DOM).

<directive
...

VGRAJENE DIREKTIVE- STRUKTURNE DIREKTIVE

- Direktiva **ngFor** je direktiva ponavljanja.
- Način, kako prikazati seznam elementov.
- Definiramo, kako naj se prikaže posamezni element, kar predstavlja predlogo za Angular pri prikazovanju vsakega elementa iz seznama.
- Primer: `<div *ngFor="let u of uporabnik">{{u.ime}}</div>`
  - Vzemi vsakega uporabnika iz seznama in ga shrani v začasno spremenljivko (**let u**) in naredi to spremenljivko dosegljivo predlogi.
- Primer.

RAZVOJ OBRAZCEV

RAZVOJ OBRAZCEV - MODEL

- Za razvoj obrazcev, prijaznih uporabnikom, potrebujemo ogrodje, ki omogoča dvosmerno vezanje podatkov, spremljanje sprememb, validacijo in upravljanje napak.
- Pred pričetkom izgradnje forme običajno opredelimo model podatkov.
  - Model posodabljamo skladno z vnosom podatkov v obrazce.
  - Model običajno predstavlja skupek lastnosti, npr.: razred Igralec.

RAZVOJ OBRAZCEV - KOMPONENTA

- Komponente za obrazce so čisto običajne komponente: vsebujejo predlogo za prikaz obrazca in pripadajoč razred, ki dela s podatki.

```
export class Film {
  constructor(public id: number,
              public naslov: string,
              public zvrst: string) { }
}
```

- Pred uporabo funkcionalnosti, ki so specifične za obrazce, moramo vključiti v app.module še `FormsModule`:
  - `import { FormsModule }   from '@angular/forms';`

RAZVOJ OBRAZCEV - PREDLOGA

- Predloga obrazca je sestavljena iz običajnih elementov in atributov HTML5.
  - Npr.: **<input>**, **required**, stili CSS...
  - Za izpis polj vrednosti si lahko pomagamo z ***ngFor**.
- Pri obrazcih običajno uporabljamo **[(ngModel)]**, saj omogoča preprosto vezanje obrazca z modelom.
  - Če uporabimo **[(ngModel)]** v obrazcu, moramo obvezno nastaviti tudi atribut HTML **name** zaradi delovanja Angular.

RAZVOJ OBRAZCEV - PREDLOGA

- Uporaba `[(ngModel)]` v obrazcu predstavlja več, kot samo dvosmerno vezanje podatkov – pove tudi, če je uporabnik spreminjal vrednosti, če je vrednost veljavna itd.
- Direktiva **ngModel** sledi spremembam in posodablja elemente s posebnimi Angular razredi.

Stanje	Razred, če drži	Razred, če ne drži
Element je bil obiskan	ng-touched	ng-untouched
Vrednost elementa se je spremenila	ng-dirty	ng-pristine
Vrednost elementa je veljavna	ng-valid	ng-invalid

RAZVOJ OBRAZCEV - PREDLOGA

- Ker ima **ngModel** vnaprej opredeljene razrede, se lahko na njih sklicujemo tudi v datoteki CSS, kjer jim nastavimo želene lastnosti.
- **ngModel** lahko tudi izvozimo v lokalno spremenljivko (**#vnosnoPolje="ngModel"**) in tako dostopamo do dodatnih lastnosti.
    - Najpogosteje se uporabljata **valid** ali **pristine** za prikaz dodatnih obvestil.
- Primer.

RAZVOJ OBRAZCEV - PREDLOGA

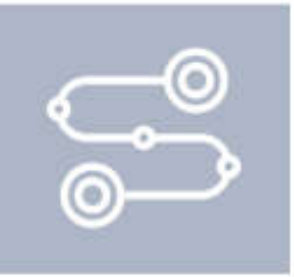
- Obrazec lahko potrdimo s pomočjo dogodka **`ngSubmit`**, ki mu nastavimo ime funkcije v komponenti.
- Obenem je tudi smiselno nastaviti referenco na **`ngForm`**, podobno, kot se naredi za **`ngModel`** na vnosnih poljih.
  - Direktiva **`ngForm`** omogoča dostop do dodatnih funkcionalnosti in nudi nadzor nad celotnim **`ngModel`** lastnostmi.
  - Ima tudi lastnost **`valid`** (veljaven), ki velja samo, ko so vsa zahtevana polja veljavna.

RAZVOJ OBRAZCEV - PREDLOGA

- Glede na veljavnost celotnega obrazca imamo tudi možnost nastavljati dodatne atribute HTML na pripadajoče elemente.
- Najpogosteje nastavljamo lastnost `disabled` na gumb za potrditev.
  - Definiramo referenčno spremenljivko na elementu `form`.
  - Na to spremenljivko se sklicujemo na gumbu.
  - Primer.

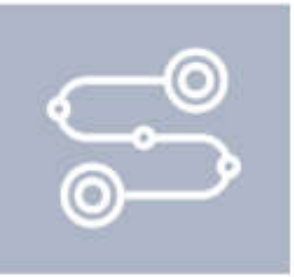
NAVIGACIJA

ROUTING



NAVIGACIJA - ROUTING

- Angular navigator (angl. Router) omogoča navigacijo med pogledi ob interakciji s strani uporabnika.
- Angular navigator temelji na navigaciji kot jo določajo brskalniki.
  - Lahko prebere URL brskalnika kot navodilo za navigacijo med odjemalskimi pogledi.
  - Lahko pošilja opcijske parametre do komponent pogledov.
  - Navigator lahko povežemo s povezavami na strani, kar omogoča da se preusmerimo do želenega aplikacijskega pogled ob kliku na povezavo.
  - Navigacijo lahko izvajamo kot rezultat različnih akcij (gumbov, menijev, kot rezultat funkcij),
  - Navigator vso aktivnost navigacije hrani v zgodovino brskalnika, kar nam omogoča prehod med pogledi s pomočjo navigacijskih tipk brskalnika.



PREDISPOZICIJA NAVIGACIJE

<base href>

- Večina aplikacij vsebuje <base> element na glavni vstopni strani (**index.html**) kot prvi element oznake **<head>**, z namenom določitve načina določanja povezave za navigatorja.

```
<base href="/">
```

@angular/router

- Angular Router je opcijska storitev, ki prikaže določen pogled komponente glede na URL.
- Ni del jedrnega Angular ogrodja.
- Pred uporabo ga je potrebno vključiti

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

KONFIGURACIJA NAVIGATORJA

appRoutes predstavlja polje poti, ki opisujejo navigacijo.

- Podamo ga metodi za konfiguracijo navigatorja, ki ga nato vključimo v korenski modul

Vsaka pot preslika URL pot v komponento.
Pri nastavljanju poti ni začetnih ali končnih znakov '/'

- Router parsa in zgradi končni URL, tako da lahko uporabljamo tako relativne kot absolutne poti.

```
const appRoutes: Routes = [
  { path: 'data', component: DataComponent },
  { path: 'data/:id', component: DataDetailsComponent },
  { path: 'home',
    component: HomeComponent,
    data: { title: "Naslov strani" }
  },
  { path: '', redirectTo: '/home', pathMatch: 'full'}
  { path: '**',
    component: NotFoundComponent },
];
export const Routing = RouterModule.forRoot(appRoutes);
```

:id nastavljen v poti predstavlja žeton za parameter poti. '/data/2' bo prebral vrednost 2 kot parameter id.

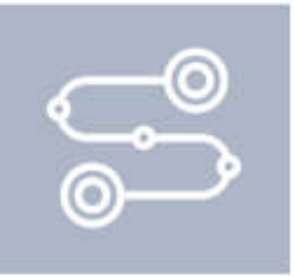
Lastnost data je mesto, kjer lahko hranimo dodatne podatke, povezane z določeno potjo. Lastnost je dostopna znotraj vsake aktivirane poti.

Prazna pot predstavlja privzeto pot aplikacije. Predstavlja mesto kamor aplikacija navigira v kolikor je URL prazen (navadno na začetku)

Pot ** predstavlja pot, ki jo navigator izbere v primeru ko URL ne ustreza nobeni izmed možnosti.

VRSTNI RED POTI JE POMEMBEN!

Navigator izbere prvo pot, ki ustreza pravilom, tako da morajo biti bolj specifične poti pred manj specifičnim.



UPORABA NAVIGATORJA

- Postavitev navigatorja na stran - **RouterOutlet**

```
<router-outlet></router-outlet>
<!-- Routed views go here -->
```

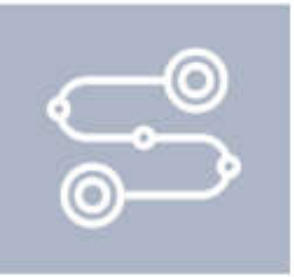
Predstavlja mesto, kjer se bo prikazala komponenta, določena s potjo.

- Uporaba povezav – **RouterLinks**

PRIMER POVEZAV

```
template: `
  <h1>Angular Router</h1>
  <nav>
    <a routerLink="/" routerLinkActive="active">Domov</a>
    <a routerLink="/data" routerLinkActive="active">Podatki</a>
  </nav>
  <router-outlet></router-outlet>`
```

- Direktiva **RouterLink** določa povezavo, kamor se navigiramo ob kliku.
- Direktiva **RouterLinkActive** omogoča vizualno razlikovati aktivnost poti.
  - Navigator doda CSS razred 'active' element, ko postane povezava, definirana v lastnosti routerLink aktivna.



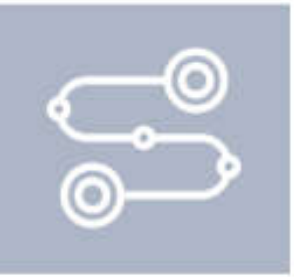
UPORABA NAVIGATORJA

- Navigacija preko programske kode – **Router**

PRIMER NAVIGACIJE

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';
…
@Component({ selector: 'komponenta',
  template: '…'})
export class TestComponent {
  constructor(private router: Router) {}
  odpriPovezavo(povezava: string) {
    this.router.navigateByUrl(povezava);
    // this.router.navigate(['povezava', parameter]);
  }}
```

- Stanje navigatorja – **RouterState**
  - Po koncu vsakega navigacijskega cikla, router zgradi objekt **ActivatedRoute**, ki vsebuje trenutno stanje navigatorja, do katerega lahko dostopamo kjerkoli v aplikaciji preko storitve **Router** in lastnosti **routerState**.



UPORABA NAVIGATORJA

- Prebiranje parametrov iz URL – **Router**

```
import { Component } from '@angular/core';
import { ActivatedRoute, Params } from '@angular/router';
…
@Component({ selector: 'komponenta', template: '…'})
export class TestComponent {
  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.params.subscribe((params: Params) => {
      console.log(params["parameter"]);
    }
}
```

PRIMER BRANJA PARAMETROV

Prebiranje route parametrov:
`this.activatedRoute.params`

Prebiranje query parametrov:
`this.activatedRoute.queryParams`

STORITVE
SERVICES

STORITVE - SERVICES

- Angular vsebuje svoj mehanizem za vrivanje odvisnosti, ki lahko nastopa tudi kot samostojen modul za ostale aplikacije in ogrodja.

- Angular Storitve

```
import { Injectable } from '@angular/core';
Import { PODATKI} from '…'; // Zbirka podatkov


@Injectable()
export class AppService {
  pridobiPodatke() { return PODATKI; }
}
```

PRIMER STORITVE

Dekorator **@Injectable** pove TypeScript-u, da le ta izpostavi metapodatke o tej storitvi.

Storitve niso nič več kot navaden razred v Angular in ostanejo to dokler jih ne registriramo z Angular injektorjem.

UPORABA STORITEV

- Injektor (angl. Injector) omogoča, da lahko storitve uporabljajo druge komponente ali storitve.
- Injektorja ni potrebno kreirati, saj Angular kreira injektor za celotno aplikacijo v času postavitve (angl. Bootstrap).

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

- Da lahko uporabimo ustvarjene storitve, je le-te potrebno registrirati med **ponudniki** (angl. providers) znotraj:
  - **NgModule** (na voljo celotni aplikaciji),
  - **Komponente** (na voljo le komponenti in njenim podkomponentam)

PRIMER PREPROSTEGA KORENSKEGA MODULA

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

UPORABA STORITEV

- Priprava komponente za vrivanje storitve:

```
import { Component } from '@angular/core';
import { AppService} from './ app.service';


@Component({ selector: 'komponenta',
  template: `
  <div *ngFor="let podatek of podatki">
    {{podatek.naziv}}
  </div>
})
export class PodatkiListComponent {
  podatki: any[];
  constructor(private appService: AppService) {
    this.podatki = appService.pridobiPodatke();
  }
}
```

KOMPONENTA KI UPORABLJA STORITEV

Tip parametra v konstruktorju, dekorator **@Component** in nastavljeni **ponudniki** nudijo dovolj informacij, da lahko Angular injektor vstavi instanco storitve AppService, vsakič ko kreiramo novo komponento **PodatkiListComponent.**

Primerek komponente bi lahko izdelali tudi implicitno (glej zgoraj), vendar je to redko potrebno saj za to poskrbi Angular sam v kolikor zazna HTML označbo komponente ali pri navigaciji do komponente preko navigatorja (angl. router).

STORITEV JE VZOREC EDINEC?

- Odvisnosti predstavljajo vzorec edinec znotraj območja injektorja.
- V našem primeru si lahko instance ene storitve AppService deli več komponent.

- Potrebno pa je vedeti, da je vrivanje odvisnosti znotraj Angular-ja hierarhično, kar pomeni, da lahko gnezdeni injektorji ustvarijo svoje nove instance.

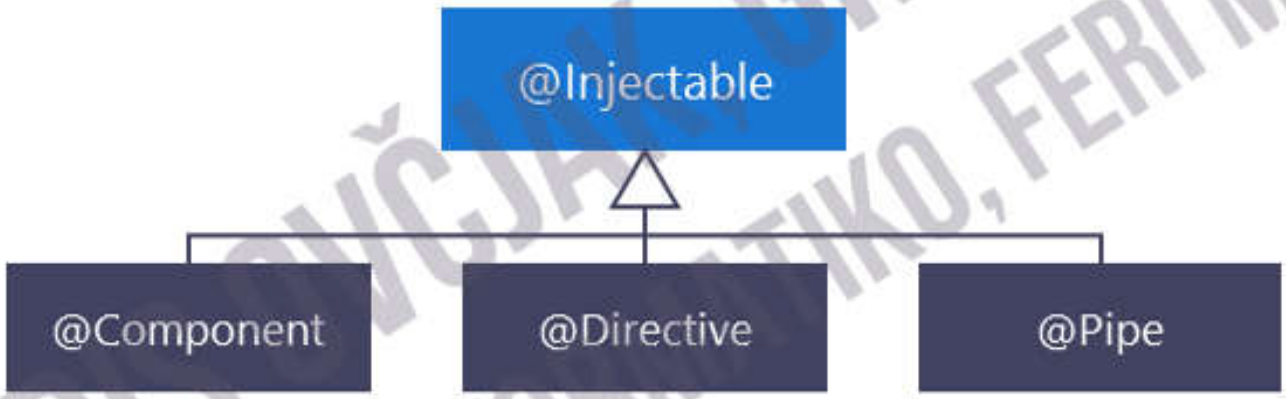
@INJECTABLE

- `@Injectable` označuje razred, ki je na voljo injektorju za instanciiranje
  - Injektor javi napako, če želimo instanciirati razred, ki ni tako označen.
- Navadno ključen del vsakega razreda, ki ga želimo vriniti

- @Injectable lahko sicer izpustimo, v kolikor razred ne vsebuje dodatnih odvisnosti, vendar je priporočljiv za vsak razred storitve zaradi:
  - **Dolgoročne varnosti (angl. Future proofing)** – ni potrebno kasneje dodajati označbe, v kolikor se odvisnost pojavi kasneje,
  - **Konsistentnosti** – Vse storitve sledijo enakim pravilom.

@INJECTABLE

- Injektorji so odgovorni tudi za instanciiranje komponent, direktiv in pip.
- Zakaj tudi omenjenim razredom ne dodamo označbe @Injectable?



Ker so vsi ti dekoratorji podtipi tipa Injectable

- Injektor se zanaša na ponudnike (angl. providers) za kreiranje instanc storitev, ki jih injektor vrine v komponente ali ostale storitve.
- Storitvene ponudnike je potrebno registrirati z injektorjem, da bo le-ta lahko ustvaril storitve.

```
Providers: [AppService]
```

HTTP

DELO S SPLETNIMI STORITVAMI

DELO S SPLETNIMI STORITVAMI

- HTTP je primarni protokol za komunikacijo med strežnikom in brskalnikom.
- Delo s HTTP storitvami je omogočeno preko modula `HttpModule`
- HTTP modul ni del jedrnega Angular modula tako, da ga je potrebno ločeno vključiti preko modula `@angular/http`.

PRIPRAVA PROJEKTA

```
import { NgModule } from '@angular/core';
import { HttpModule } from '@angular/http';

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

DELO S SPLETNIMI STORITVAMI

- Primer preprostega klica GET metode za pridobivanje podatkov

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/map';

@Injectable()
export class AppService {
  private url = "api/podatki";
  constructor(private http: Http) {}

  pridobiPodatke(): Observable<Podatek[]> {
     return this.http.get(this.url)
        .map(this.extractData)
        .catch(this.handleError);
  } … }
```

Uporaba knjižnice RxJS

Angular Http odjemalca vrinemo v konstruktor storitve

Klic storitve GET (uporaba RxJS)

- Metoda vrne **Observable<Response>** ki ga obdelamo z **map**, napake pa lovimo z operatorjem **catch**.
- Response lahko pretvorimo tudi v Promise v kolikor želimo.

KNJIŽNICA RXJS

- Gre za zunanjo knjižnico, ki jo podpira Angular.
- Namenjena delu s http poizvedbami in implementira vzorec asinhronega opazovanja (angl. asynchronous observable),

- Vključitev knjižnice

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/map';
```

OBDELAVA ODGOVORA STORITVE

- Pri obdelavi podatkov nam storitev vrne odgovor tipa **Response**, ki ga lahko obdelamo v želen tip.
- Pretvorba se zgodi znotraj operatorja **map**.
  - Lahko kličemo zunanjo metodo
  - Pretvorbo izvedemo v gnezdenem stavku

PRIMER ZUNANJE METODE

```
private extractData(res: Response) {
  let body = res.json();
  return body.data || {};
}
```

PRIMER GNEZDENEGA STAVKA

Parsanje JSON objekta v Javascript objekt

```
return this.http.get(this.url)
        .map((res: Response) => {
          let body = res.json();
          return body.data || {};
        })
        .catch(this.handleError);
```

OBRAVNAVANJE NAPAK

- Pomemben del klica storitev je tudi pričakovanje napak in priprava na lovljenje le-teh.
- Napake lovimo z operatorjem **catch**, kjer lahko definiramo metodo, ki bo napako obravnavala.
- Metoda preoblikuje napako v objekt, ki ga lahko obdelamo.

PRIMER OBRAVNAVE NAPAK

```
private handleError(error: Response | any) {
  let errMsg: string;
  if(error instanceof Response) {
    let body = error.json() || '';
    const err = body.error || JSON.stringify(body);
    errMsg = '${error.statusText}';
  }
  else { errMsg = error.message ? error.message : error.toString() }
  return Observable.throw(errMsg);
}
```

KOMPONENTA IN KLIC STORITVE

- Klic storitve se začne v komponenti, kjer uporabnik preko metode pokliče spletno storitev, ki se nahaja v Angular storitvi.

PRIMER KLICA

```
pridobiPodatke() {
  this.appService.pridobiPodatke()
      .subscribe(
          podatki => this.podatki = podatki,
          napaka => this.sporociloNapake = <any>napaka);
}
```

- V komponenti uporabimo vrinjen razred storitve za klic storitve.
- S funkcijo **subscribe** pridobimo podatke, ki nam jih vrne storitev, prav tako pa lahko obravnavamo morebitne napake pri klicu storitve.
  - Funkcija subscribe vsebuje drugi funkcijski parameter, namenjen obravnavanju napak.

POŠILJANJE PODATKOV

- Klic storitve se začne v komponenti, kjer uporabnik preko metode pokliče spletno storitev, ki se nahaja v Angular storitvi.

```
import { Headers, RequestOptions } from '@angular/http'
…

dodajPodatek(d: string): Observable<Podatek> {
  let headers = new Headers(
      { 'Content-Type': 'application/json'});
  let options = new RequestOptions({ headers: headers });

  return this.http.post(this.url, JSON.stringify(d), options)
     .map(this.extractData)
     .catch(this.handleError);
}
```

PRIMER KLICA IZ STORITVE

Objekt **headers** uporabimo za določitev možnosti (options) - **RequestOptions** pri klicu storitve.
Objekt predstavlja tretji parameter pri klicu storitve

Obravnava odgovora poizvedbe je enaka ko pri ostalih klicih (GET, PUT, DELETE)

ANGULAR 2

ALI 4 ...?

ANGULAR 4.X

- NEVIDNA PREOBRAZBA - INVISIBLE MAKEOVER -

- Semantično verzioniranje
  - Verzije ne bodo spreminjale funkcionalnosti, manjše verzije bodo vsebovale le manjše dodane spremembe, večje spremembe bodo rezervirane za večje izide.
  - Verzioniranje vezano na čas (6 mesečni cikel)
- Povratno kompatibilen z verzijo 2.x.x za večino aplikacij

NOVOSTI

Manjši in hitrejši

- Velikost generirane kode komponent zmanjšana do 60%
- Animacije ločene v svojem paketu (projekti, ki ne vsebujejo animacij je ne potrebujejo vključevati)

ANGULAR 4.X

NOVOSTI

Nove funkcionalnosti

- Izboljšani direktivi *ngIf in *ngFor
  - Sintaksa predlog sedaj vsebuje možnost if/else stila sintakse, prav tako pa določanje lokalnih spremenljivk.

```
<div *ngIf="userList | async as users; else loading">
  <user-profile *ngFor="let user of users; count as count; index as i" [user]="user">
User {{i}} of {{count}}
  </user-profile>
</div>
<ng-template #loading>Loading...</ng-template>
```

- Angular Universal
  - Omogoča izvajanje Angular ogrodja na strežniku
  - Za uporabnika predstavlja minimalno spremembo (200ms razlike v odzivnem času)

ANGULAR 4.X

NOVOSTI

Nove funkcionalnosti

- Kompatibilnost s TypeScript 2.1 in 2.2
- Mapiranje vira za predloge
  - V primeru napak omogoča podajanje natančnejšega konteksta v povezavi z izvorno predlogo.

Spremembe paketiranja (angl. packaging)

- Sploščeni (angl. Flat) ES moduli (Flat ESM / FESM)
  - Angular sedaj podpira sploščene verzije modulov v EcmaScript Module formatu, ki omogoča zmanjšanje velikosti generiranih paketov, hitrejše grajenje, prevajanje in nalaganje.
- Eksperimentalna kompatibilnost s Closure anotacijami

Hvala za pozornost

Boris Ovčjak

boris.ovcjak@gmail.com

boris.ovcjak@um.si

Gregor Jošt

jost.gregor@gmail.com

gregor.jost@um.si