Životni ciklus komponente

Konstruktor komponente

- Svaka komponenta ima životni ciklus i prolazi kroz različite faze od inicijalizacije do destrukcije
- Pošto je komponenta TypeScript klasa, svaka komponenta ima konstruktor
- Konstruktor komponente se prvi poziva pre poziva bilo kog događaja iz životnog ciklusa komponente
- Kod Angular aplikacija konstruktor se uglavnom koristi za ubacivanje zavisnosti (*dependency injection*)

Životni ciklus komponente - metode

- **ngOnChanges** metoda se poziva u child komponenti kada parent komponenta izmeni vrednost ulaznog svojstva označenog dekoratorom @Input()
  - Ako komponenta nema ulazna svojstva (@Input), metoda ngOnChanges se ne poziva
- **ngOnInit** metoda se poziva kada se komponenta inicijalizuje prvi put
  - Poziva se posle konstruktora i prvog poziva metode ngOnChanges (ako postoji)
  - U trenutku poziva ove metode svojstva komponente su inicijalizovana
  - Učitavanje podataka neophodnih za prikaz komponente najčešće se obavlja unutar ove metode
- **ngDoCheck** metoda se poziva prilikom svakog prolaska Angular mehanizma za detekciju promena
  - Koristi se za ručnu detekciju i praćenje promena koje Angular automatski ne detektuje
- **ngOnDestroy** metoda se poziva neposredno pre nego što Angular uništi instancu komponente

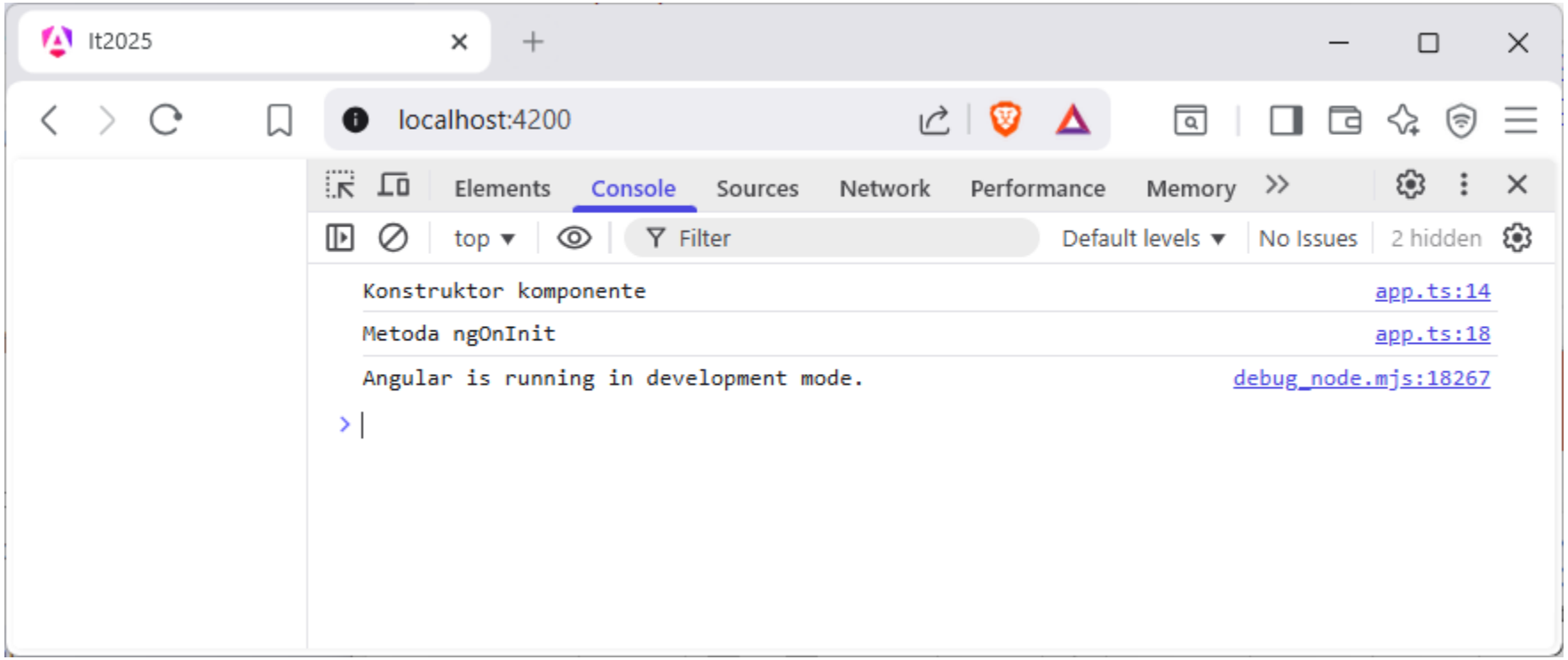
Interfejs OnInit

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class App implements OnInit {

  ngOnInit(): void {
    throw new Error('Method not implemented.');
  }

}
```

ngOnInit

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

@Component({
  selector: 'app-root',
  templateUrl: './on-init.component.html',
  styleUrls: ['./on-init.component.css']
})
export class App implements OnInit {

  title = 'komunikacija';

constructor() {
  console.log('Konstruktor komponente');
}

ngOnInit(): void {
  console.log("Metoda ngOnInit");
}

}
```

ngOnInit


It2025
localhost:4200
Elements
Console
Sources
Network
Performance
Memory
top
Filter
Default levels
No Issues
2 hidden
Konstruktor komponente
app.ts:14
Metoda ngOnInit
app.ts:18
Angular is running in development mode.
debug_node.mjs:18267

Parent komponenta

```
@Component({
  selector: 'app-parent-component',
  imports: [],
  templateUrl: './parent-component.html',
  styleUrl: './parent-component.css',
})
export class ParentComponent {
  x = 0;


  promenaParent(): void {
    this.x++;
  }


}
```

```
ng g c parentComponent
```

Child komponenta

```
ng g c childComponent
```

```
export class ChildComponent {

@Input() parentData: number = 0;

constructor() {}

ngOnChanges(changes: SimpleChanges): void {
  console.log('ngOnChanges', changes);
}


ngOnInit(): void {
  console.log('ngOnInit');
}

promenaChild(): void {
  this.parentData--;
}
}
```

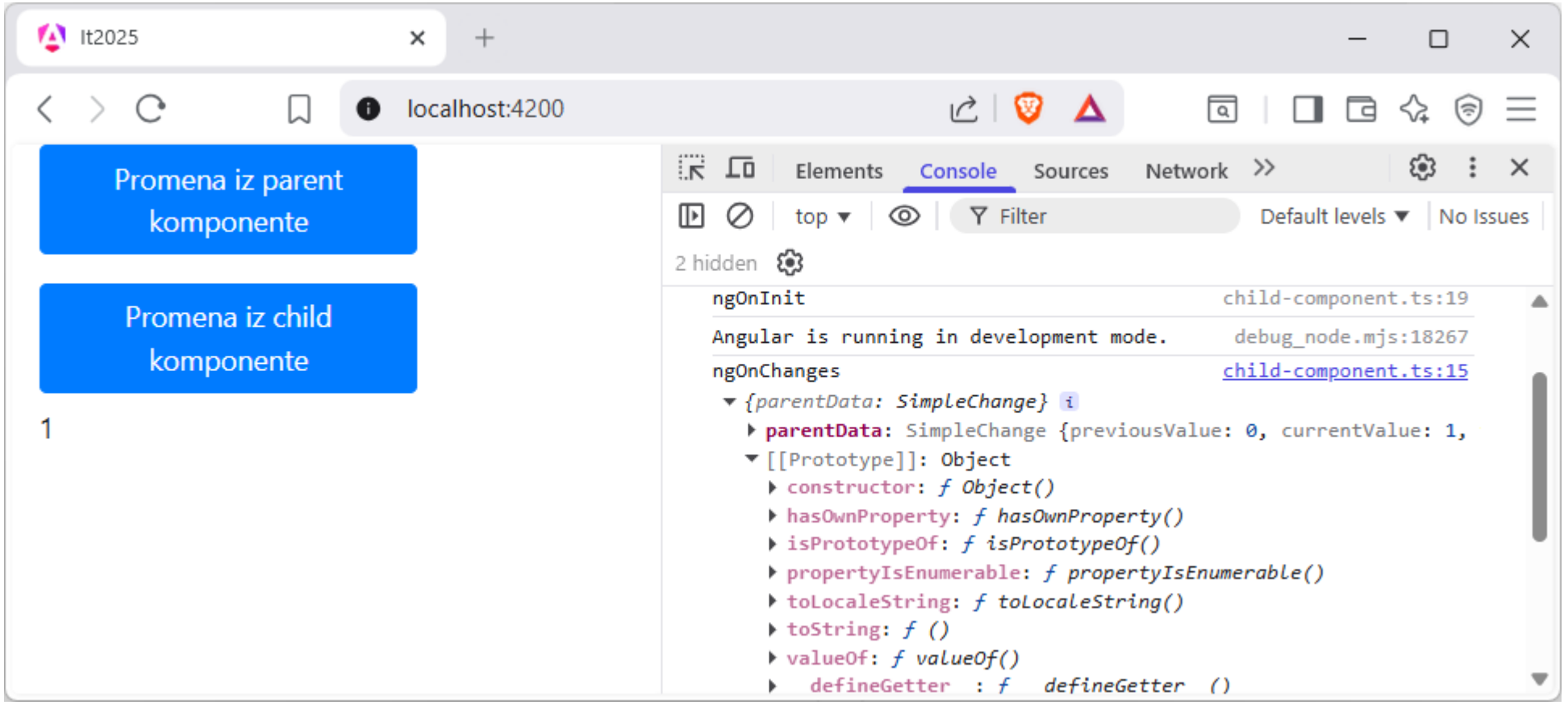
Šablon child komponente

```
<button
  class="btn btn-primary btn-block"
  (click)="promenaChild()">
  Promena iz child komponente
</button>


<div class="mt-2">
  {{ parentData }}
</div>
```

Šablon parent komponente

```
<div class="row">
  <div class="col-8">
    <button
      class="btn btn-primary btn-block"
      (click)="promenaParent()">
      Promena iz parent komponente
    </button>
  </div>
</div>


<div class="row mt-3">
  <div class="col-8">
    <app-child-component [parentData]="x"></app-child-component>
  </div>
</div>
```

Modifikacija ulaznog svojstva posredstvom parent komponente



Angular cevi (pipes)

- Omogućavaju transformaciju podataka pre njihovog prikaza u pogledu
- Potrebno je importovati `CommonModule`
- Pipe karakter se označava sa |

```
export class CeviComponent {
 naslov = "Angular cevi";
 datum:Date = new Date();
 a:number = 5.678;
 p:number = 0.257;
}
```

Rad sa stringovima

```
<div class="row">
<h3>{{ naslov }}</h3>
<h3>{{ naslov | lowercase }}</h3>
<h3>{{ naslov | uppercase }}</h3>
<h3>{{ naslov | titlecase }}</h3>
<h2>{{ naslov | slice:3 }}</h2>
<h2>{{ naslov | slice:3:6 }}</h2>
</div>
```

Angular cevi

angular cevi

ANGULAR CEVI

Angular Cevi

ular cevi

ula

Rad sa datumom

```
<p>shortDate: {{ datum | date:'shortDate' }}</p>
<p>shortTime: {{ datum | date:'shortTime' }}</p>


<p>dd.MM.yyyy: {{ datum | date:'dd.MM.yyyy' }}</p>
<p>d.M.yyyy: {{ datum | date:'d.M.yyyy' }}</p>


<p>HH:mm:ss: {{ datum | date:'HH:mm:ss' }}</p>
```

shortDate: 12/9/25

shortTime: 10:21 AM

dd.MM.yyyy: 09.12.2025

d.M.yyyy: 9.12.2025

HH:mm:ss: 10:21:37

Rad sa brojevima

```
<p>{{ a | number:'1.2-3' }}</p>
<p>{{ a | number:'3.4-5' }}</p>
<p>{{ a | number:'3.1-2' }}</p>
```

5.678

005.6780

005.68

Rad sa procentima

```
<div>{{ p | percent }}</div>
<div>{{ p | percent:'2.3-5' }}</div>
```

26%
25.700%

Servisi

Servisi

- **Komponenta** je fokusirana na interaktivnu logiku sa korisnikom
- Preuzimanje podataka sa servera ne treba da se obavlja u komponenti
- **Servis** je klasa koja komunicira sa serverom i obezbeđuje podatke aplikaciji
- Servis je klasa sa uskom i jasno definisanom funkcionalnošću
- Komponenta koristi servis putem mehanizma koji se naziva *dependency injection* (DI)
- Pomoću anotacije @**Injectable**() definiše se da je servis dostupan za dependency injection

Dependency Injection u Angularu

- Angular poseduje sopstveni **DI** framework
- **Zavisnosti** su servisi ili objekti koji su potrebni nekoj klasi da bi izvršavala svoju funkcionalnost
- **Dependency Injection** (DI) je dizajnerski obrazac u kome klasa dobija svoje zavisnosti iz spoljnog izvora, umesto da ih sama instancira
- Dobavljanje zavisnosti od DI framework-a vrši se posredstvom konstruktora zavisne klase

Kreiranje klase

ng g class osoba

```
export class Osoba {
    constructor(public id: number, public ime: string, public prezime: string, public starost: number) {
    }
}
```

Fajl podaci.ts unutar app foldera

```
import { Osoba } from './osoba';

export const OSOBE: Osoba[] = [
    new Osoba(1, 'Marko', 'Markovic', 25),
    new Osoba(2, 'Petar', 'Petrovic', 34),
    new Osoba(3, 'Jovan', 'Jovanovic', 27),
    new Osoba(4, 'Milan', 'Mirkovic', 22)
  ];
```

Kreiranje servisa

```
ng g service osbaService
```

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


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

}
```

Singleton servis je servis za koji postoji **jedna instanca u aplikaciji**.
Opcija providedIn: 'root' obezbeđuje da servis bude dostupan na nivou **cele aplikacije** i da postoji **jedinstvena instanca** koja se deli između svih komponenti i drugih servisa.

Servis OsobaService

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


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

  }
  vratiOsobe(){
    return OSOBE;
  }
}
```

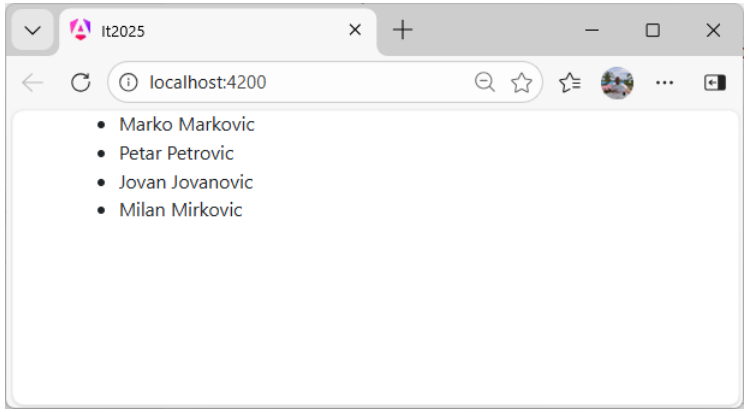
Ubacivanje objekta servisa u konstruktor komponente

```
ng g c osobe-component
```

```
export class OsobeComponent implements OnInit {
  title = 'Servisi';
  osobe: Osoba[] =[];


  constructor(private oServis: OsobaService) { }


  ngOnInit(): void {
    this.osobe = this.oServis.vratiOsobe();
  }
}
```

Šablon komponente

```
<div class="row">
  <div class="col-6">
    <ul>
      @for (osoba of osobe; track osoba) {
        <li>
          {{ osoba.ime }} {{ osoba.prezime}}
        </li>
      }
    </ul>
  </div>
</div>
```



Observables

- **Observable** je objekat koji predstavlja tok podataka i može emitovati niz vrednosti tokom vremena
- **Observable** može emitovati vrednosti asinhrono
- Vrednosti koje emituje Observable se šalju *observerima* koji pratetaj Observable
- Kada se **Observer** pretplati(*subscribe*) na Observable, on postaje "slušalac" koji reaguje na:
  - emitovanu vrednost
  - grešku
  - obaveštenje o završetku toka podataka

Observer

- **Observer** je objekat koji definiše tri moguće metode: next, error i complete
- Metoda next se poziva kada **Observable** emituje novu vrednost
  - Ova metoda prima vrednost koja je upravo emitovana
- Metoda error se poziva kada **Observable** emituje grešku
  - Ova metoda prima objekat koji predstavlja grešku
- Metoda complete se poziva kada **Observable** završi emitovanje vrednosti

Observables

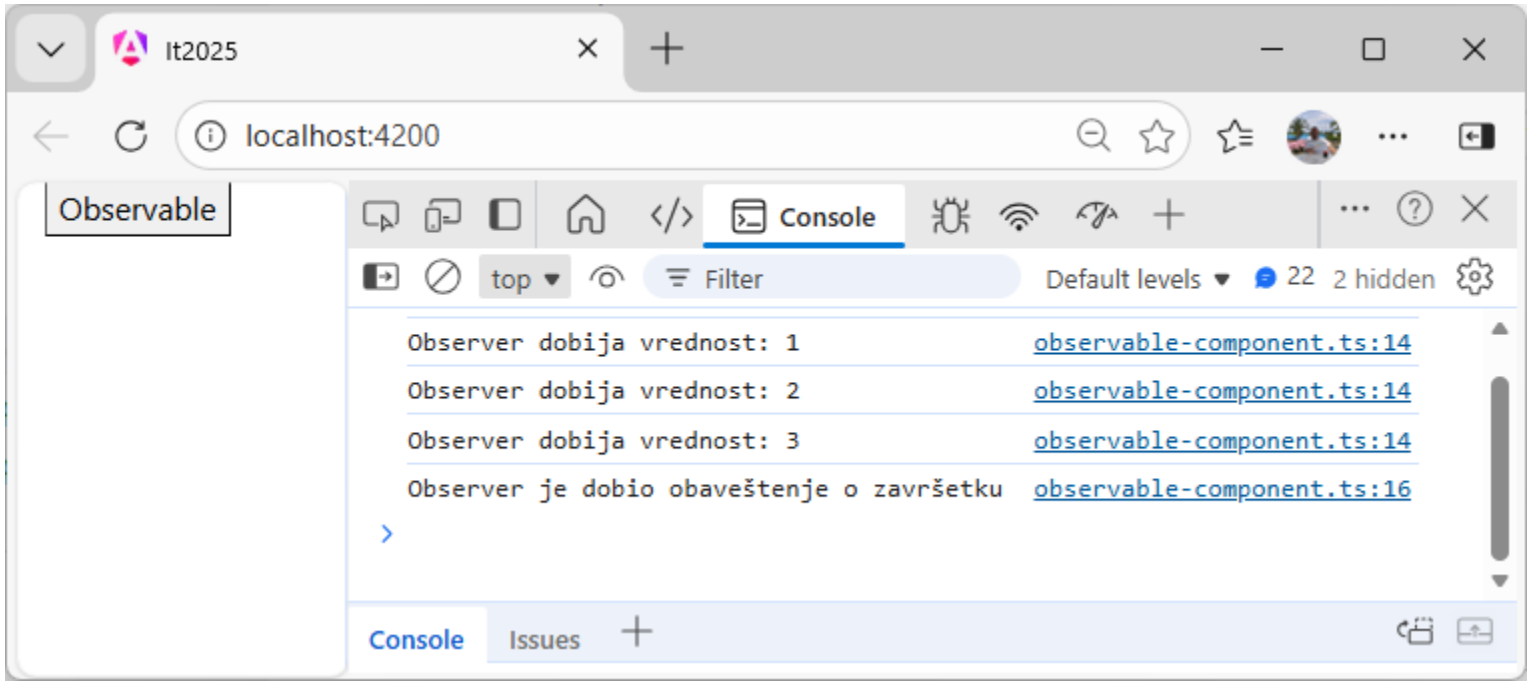
- Web server obično šalje podatke asinhrono i ti podaci se mogu modelovati kao Observable tok (stream)
- Observable predstavlja sekvencu vrednosti koje pristižu asinhrono tokom vremena
- Observable može imati više pretplatnika i svi pretplatnici bivaju obavešteni kada se stanje Observable-a promeni
- Angular koristi biblioteku **RxJS (Reactive Extensions for JavaScript)** za rad sa Observable-ima
- **HTTP modul** u Angularu koristi Observable-e prilikom obrade AJAX zahteva i odgovora

Primer Observable i pretplata

```
export class ObservableComponent {
mojObservable = of(1, 2, 3);


  observer = {
    next: (x: number) => console.log('Observer dobija vrednost: ' + x),
    error: (err: string) => console.error('Observer je dobio grešku: ' + err),
    complete: () => console.log('Observer je dobio obaveštenje o završetku')
  };


  pretplata() {
    this.mojObservable.subscribe(this.observer);
  }
}
```

Šablon komponente

```
<button (click)="pretplata()">Observable</button>
```



Modifikacija servisa

```
@Injectable({
  providedIn: 'root',
})
export class OsobaService {
  constructor() {

  }
  // vratiOsobe(){
  //   return OSOBE;
  // }
  vratiOsobe(): Observable<Osoba[]> {
    return of(OSOBE);
  }
}
```

Pretplata na observable – prosleđivanje metode

```
export class OsobeComponent implements OnInit {
  title = 'Servisi';
  osobe: Osoba[] =[];


  constructor(private oServis: OsobaService) { }


  ngOnInit(): void {
    //this.osobe = this.oServis.vratiOsobe();
    this.oServis.vratiOsobe()
      .subscribe(os => this.osobe = os);

  }
}
```

Komunikacija sa udaljenim serverom

Angular HttpClient servis

- **HttpClient** je servis koji se koristi za slanje HTTP zahteva i prijem odgovora
- Omogućava asinhronu komunikaciju sa udaljenim serverom
- Nalazi se u paketu @angular/common/http
- U standalone Angular aplikacijama koristi se funkcija **provideHttpClient**() za registrovanje HttpClient servisa u aplikaciji
- Funkcija provideHttpClient() se poziva prilikom pokretanja(**bootstrapanje)** aplikacije
- **Bootstrapanje** aplikacije je proces pokretanja i inicijalizacije Angular aplikacije i vrši se u fajlu main.ts pomoću funkcije **bootstrapApplication**()

Obezbeđivanje HttpClient servisa u angular aplikaciji

Fajl main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { App } from './app/app';
import { provideHttpClient } from '@angular/common/http';


// bootstrapApplication(App, appConfig)
//   .catch((err) => console.error(err));


bootstrapApplication(App, {
  providers: [
    provideHttpClient()
  ]
});
```

Servis HttpClient

- Klasa **HttpClient** je tzv. injectable klasa i ubacuje se u servis putem njegovog konstruktora
- Klasa HttpClient sadrži metode kojima se šalju HTTP zahtevi ka serveru
- Podatke za aplikaciju najčešće obezbeđuje **Web API**
- Metoda **get**() šalje GET zahtev (čitanje podataka)
- Metoda **post**() šalje POST zahtev (kreiranje novih podataka)
- Metoda **put**() šalje PUT zahtev (izmena postojećih podataka)
- Metoda **delete**() šalje DELETE zahtev (brisanje postojećih podataka)

Angular In-Memory Web API

- Ovaj modul omogućava simulaciju REST API back-enda u memoriji, prvenstveno za razvoj i testiranje aplikacija bez pravog servera

```
npm i angular-in-memory-web-api
```

Kreiranje baze podataka u memoriji

ng g class baza

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
import { Osoba } from './osoba';

export class Baza implements InMemoryDbService {
    createDb() {
      const osobe: Osoba[] = [
                new Osoba(1, 'Marko', 'Markovic', 25),
                new Osoba(2, 'Petar', 'Petrovic', 34),
                new Osoba(3, 'Jovan', 'Jovanovic', 27),
                new Osoba(4, 'Milan', 'Mirkovic', 22)
              ];

      return {osobe};
    }
}
```

'api/osobe' adresa web api -ja

Konfigurisanje memorijskog web apija

```
bootstrapApplication(App, {
  providers: [
    provideHttpClient(),
    importProvidersFrom(InMemoryWebApiModule.forRoot(Baza))
  ]
});
```

Kreiranje servisa za komunikaciju sa api -jem

```
import { Injectable } from '@angular/core';
import { throwError } from 'rxjs';
import { Osoba } from './osoba';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';


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

  private apiUrl = 'api/osobe';


  constructor(private http: HttpClient) {}


  private errorHandler(error: HttpErrorResponse) {
    return throwError(() => error);
  }
}
```

Generisanje GET zahteva ka Web serveru

- Komponenta poziva GET metodu Angular servisa
- Servis koristi **HttpClient** za slanje **HTTP** zahteva
- Kreira se **HTTP GET** zahtev ka Web API ruti
- Server obrađuje zahtev i vraća odgovor
- Odgovor sadrži podatke u **JSON** formatu
- Angular prima JSON i mapira ga u odgovarajuće objekte
- Rezultat se prosleđuje komponenti kroz **Observable**

Slanje podataka GET zahtevom

- GET metoda ne šalje podatke u telu zahteva
- Podaci se prosleđuju isključivo kroz URL adresu
- GET metoda se koristi za preuzimanje(čitanje) podataka sa servera
- Načini prosleđivanja podataka:
  - **Putanja (path parametri)** predstavlja deo URL adrese koji se mapira na odgovarajuću serversku klasu (kontroler) i njenu metodu
    - U primeru GET /osobe/5, deo **/osobe** označava serverski kontroler zadužen za rad sa osobama, dok broj **5** predstavlja **ID** resursa koji se prosleđuje metodi tog kontrolera
  - **Query parametri** se koriste za slanje dodatnih kriterijuma
    - Primer: GET /osobe?grad=Beograd&starost=25

GET metode servisa za komunikaciju sa Web Api aplikacijom

```
vratiOsobe(): Observable<Osoba[]> {
    return this.http.get<Osoba[]>(this.apiUrl)
      .pipe(catchError(this.errorHandler));
  }

  vratiOsobu(id: number): Observable<Osoba> {
    const url = `${this.apiUrl}/${id}`;
    return this.http.get<Osoba>(url).pipe(
      catchError(this.errorHandler)
    );
  }
```

pipe() funkcija prima operatore (funkcije) koje se ulančavaju nad Observable-om
catchError() je operator koji hvata greške i prosleđuje ih dalje preko errorHandler metode

Obrada grešaka u http komunikaciji

```
private errorHandler(error: HttpErrorResponse) {
  // Logovanje greške (za developere)
  console.error('Došlo je do greške:', error);

  let errorMessage = 'Nepoznata greška';

  if (error.error instanceof ErrorEvent) {
    // Greška na klijentskoj strani
    errorMessage = `Klijentska greška: ${error.error.message}`;
  } else {
    // Greška na serverskoj strani
    errorMessage = `Serverska greška (${error.status}): ${error.message}`;
  }

  // Vraćanje greške kao Observable
  return throwError(() => new Error(errorMessage));
}
```

throwError funkcija iz RxJS biblioteke kreira Observable koji će emitovati grešku
Anonimna funkcija vraća samu grešku kao vrednost koja će biti emitovana kroz Observable.

Komponenta – ubacivanje objekta servisa

```
export class ListaOsobaComponent {
  osobe: Osoba[] = [];
  odabranaOsoba: Osoba = new Osoba(0, '', '', 0);
  idOsobe = 0;
  imeOsobe = "";
  prezimeOsobe = "";
  starostOsobe = 0;
  resetujPolja(): void {
    this.imeOsobe = '';
    this.prezimeOsobe = '';
    this.starostOsobe = 0;
  }


  constructor(private oServis: OsobaService) {
  }
```

Metode komponente: prikaziOsobe() i prikaziOsobu()

```
prikaziOsobe(): void {
  this.oServis.vratiOsobe()
    .subscribe({
      next: os => this.osobe = os,
      error: grr => console.error('Greska: ' + grr)
    });
}
```

```
prikaziOsobu(): void {
  this.oServis.vratiOsobu(this.idOsobe)
    .subscribe({
      next: os => this.odabranaOsoba = os,
      error: grr => {
        console.log('Greska: ' + grr);
        this.odabranaOsoba = new Osoba(0, '', '', 0);
      }
    });
}
```

Metode komponente za prikaz podataka

- Metoda **prikaziOsobe**() poziva servis i šalje HTTP GET zahtev za preuzimanje liste osoba
- Komponenta se pretplaćuje (subscribe) na **Observable** koji vraća servis
- Nakon uspešnog odgovora, lista osoba se smešta u niz osobe i prikazuje u šablonu
- Metoda prikaziOsobu() šalje HTTP GET zahtev sa ID-jem osobe
- Sa servera se preuzima jedna osoba i smešta u objekat odabranaOsoba
- U slučaju greške, greška se obrađuje u error delu subscribe metode

Metoda ngOnInit()

```
ngOnInit(): void {
    this.prikaziOsobe();
  }
```

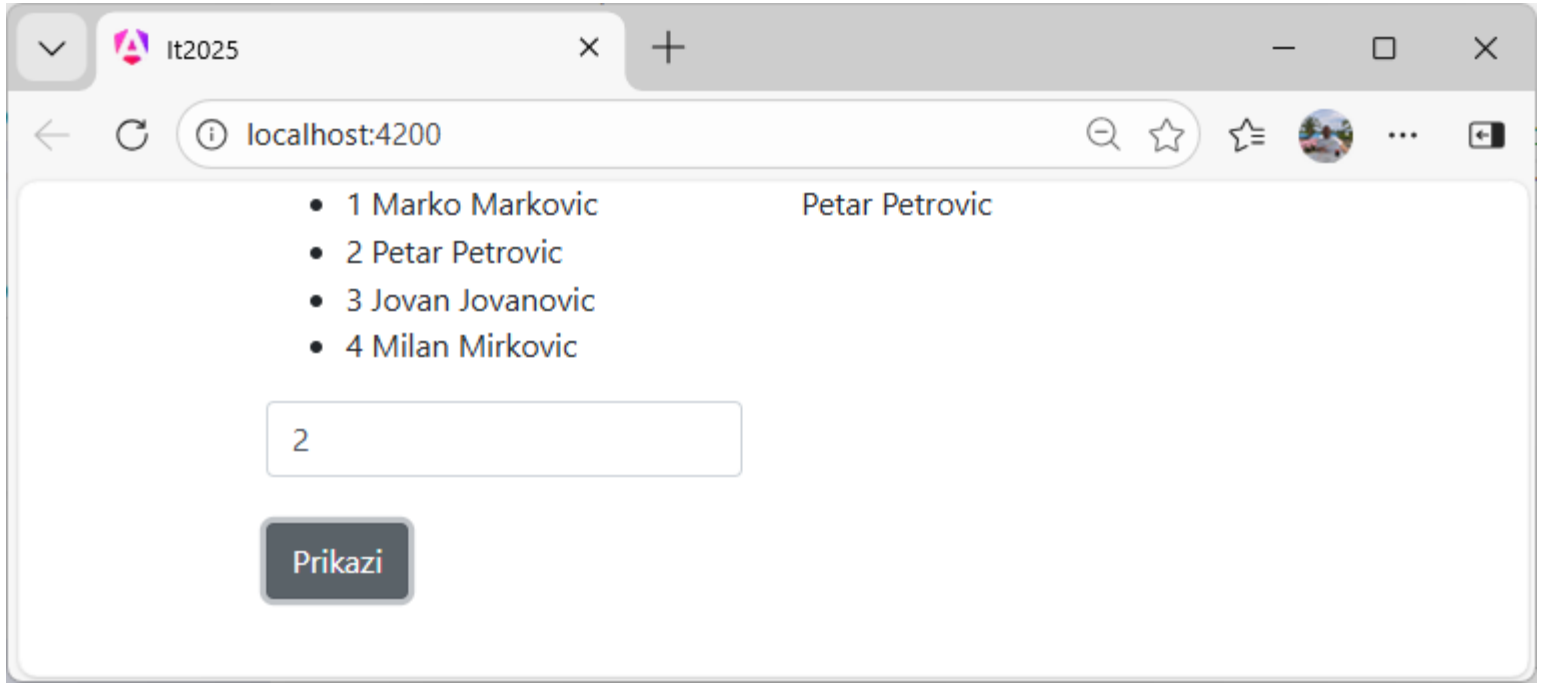
Šablon komponente za prikaz podataka

```
<div class="row">
  <div class="col-6">
    <ul>
      @for (osoba of osobe; track osoba.id) {
        <li>
          {{ osoba.id }} {{ osoba.ime }} {{ osoba.prezime }}
        </li>
      }
    </ul>
  </div>
  @if (odabranaOsoba.id !== 0) {
    <div class="col-6">
      <p>
        {{ odabranaOsoba.ime }} {{ odabranaOsoba.prezime }}
      </p>
    </div>
  }
</div>
```

Šablon komponente za unos podataka

```
<div class="row">
    <div class="col-6">
        <div class="form-group" >
            <input type="text" [(ngModel)]="idOsobe" placeholder="Unesi ID osobe"
            class="form-control"> <br>
            <button class="btn btn-secondary“
            (click)="prikaziOsobu()">Prikazi</button> <br>
        </div>
    </div>
</div>
```

Korisnički interfejs



Pitanje 1

Šta se prvo izvršava prilikom kreiranja Angular komponente?

a. konstruktor
b. ngOnInit
c. ngDoCheck

Odgovor: a

Pitanje 2

Metoda ngOnChanges se poziva:

a. u parent komponenti kada se promene njeni lokalni property
b. u child komponenti kada parent komponenta modifikuje ulazni property u child komponenti
c. u child komponenti kada ona sama promeni vrednost svog ulaznog property-ja

Odgovor: b

Pitanje 3

Angular cevi (pipes) se koriste za:

a. komunikaciju sa udaljenim serverom
b. razmenu podataka između komponenti
c. transformaciju podataka pre njihovog prikaza u pogledu

Odgovor: c

Pitanje 4

Angular klasa koja je zadužena za komunikaciju sa serverom i obezbeđivanje podataka aplikaciji naziva se:

a. servis
b. pipe
c. komponenta

Odgovor: a

Pitanje 5

Angular servis se označava dekoratorom:

a. @Service
b. @HttpService
c. @Injectable

Odgovor: c

Pitanje 6

Ubacivanje objekta servisa u klasu komponente vrši se :

a. u metodi ngOnInit komponente
b. u konstruktoru komponente
c. u metodi ngDoCheck komponente

Odgovor: b

Pitanje 7

Klasa koja omogućava komunikaciju sa udaljenim Web API-jem naziva se:

a. HttpClient
b. HttpModule
c. HttpConnect

Odgovor: a

Pitanje 8

Da bi se koristila klasa HttpClient za komunikaciju sa udaljenim serverom u standalone aplikaciji u Angular 20, potrebno je:

a. koristiti funkciju **provideHttpClient**() pri pokretanju aplikacije (u funkciji bootstrapApplication)
b. Importovati **HttpClientModule** u svakom servisu koji koristi **HttpClient**
c. Imortovati **RemoteModule** u svakom servisu koji koristi **HttpClient**

Odgovor: a