5. Angular i RxJS Pipes

Sadržaj

- Imperativno i reaktivno programiranje
- Biblioteka RxJS
- Reaktivno programiranje u Angularu
- Pipes

Ukratko o reaktivnom programiranju

- Razumevanje osnovnih koncepata reaktivnog programiranja je potrebno da bismo ih mogli primeniti u Angular aplikacijama
- Pogodni su za realizaciju asinhrone komunikacije Angular aplikacije se *backend* delom aplikacije
- Angular koristi RxJS biblioteku

Tokovi (engl. *streams*)

- Tokovi su sekvence vrednosti koji se javljaju tokom vremena
- Primer toka je broj koji se uvećava za 1 nakon svake sekunde

  `[0, 1, 2, 3, 4]`
- Drugi primer toka bi mogle biti x i y koordinate pozicija klika

  `[(12,34), (345,22), (1,993)]`
- Tok može da predstavlja svaki pritisak na taster

  `["D", "j", "o", "r", "d", "j", "e"]`
- Tok može da predstavlja reprezentacija upisa podataka korisnika u formu

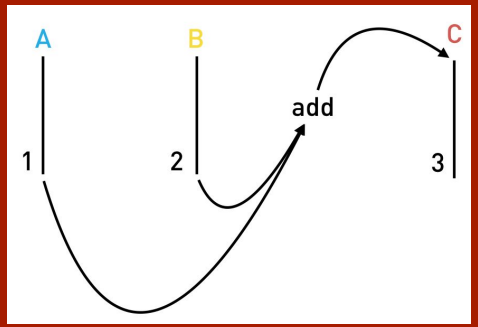
  `["D", "Dj", "Djo", "Djor", "Djord", "Djordj", "Djordje"]`

Ideja reaktivnog programiranja

- Kompletan program se može napraviti definisanjem različitih tokova i operacija koje se izvršavaju nad tim tokovima
- Ideja reaktivnog nasuprot imperativnom programiranju je drugačija po načinu posmatranja sistema

```
add ( A , B ) {
  return A + B;
}

C = add(1, 2);
C = add(1, 4);
```



Imperativno programiranje

- Poziv funkcije `add` menja stanje promenljive `C` tako što joj dodeljuje vrednost izraza `A + B`
- Kada se vrednost `B` promeni moramo naći način da “saznamo” da se to desilo
- Potom je potrebno da ponovo izračunamo `A + B` i tu vrednost dodelimo `C`
- U aplikacijama se ulazi stalno menjaju tokom vremena, najčešće kao posledica interakcije korisnika
- Većina logike aplikacije se svede na odlučivanje koje funkcije treba pozvati za određene promene ulaza

Reaktivno programiranje

- U reaktivnom programiranju ne mislimo o promenljivima već o tokovima i kako su ti tokovi međusobno povezani
- U našem primeru `A` nije promenljiva koja ima određenu vrednost u jednom trenutku vremena već tok različitih vrednosti tokom vremena
- Funkcija `add` je operacija koja povezuje izlaz tokova `A` i `B` sa ulazom toka `C`
- Kada se pojave neke vrednosti u tokovima `A` i `B`, operacija `add` se **automatski** poziva, izračunava zbir i šalje ga na tok `C`
- Ako je tok `C` dalje povezan sa drugim tokom putem neke operacije, ta operacija će se isto tako automatski izvršiti

Koncept `Observable`

- `Observable` je primitivni tip koji služi kao model za
  - kreiranje tokova
  - prijavljivanje na tokove
  - reagovanje na pojavu novih vrednosti na toku i
  - kombinovanje tokova i pravljenje novih
- Dok ne postane deo Java Script verzije (kandidat za ES7) potrebno je koristiti biblioteku RxJS koja nam omogućava korišćenje `Observable` primitiva

`Observable` u Angular aplikaciji

- Kreiramo promenljivu tipa `Observable`
- Poziv `interval(1000)` svakih 1000 milisekundi generiše sledeći celi broj
- `Observable` je *cold* dok se neko ne prijavi na njega, nakon čega postaje *hot*
- Fat arrow funkcija ispisuje na konzolu poruku koja sadrži generisanu vrednost

```
import { interval } from 'rxjs';
...

@Component(...)
export class AppComponent implements OnInit {
  ...
  ngOnInit() {
    const obs = interval(1000);
    obs.subscribe(value => console.log('Subscriber '
                                       + value));
  }
}
```

callback - kod koji se izvršava pri pojavi vrednosti na toku

http://reactivex.io/documentation/operators/interval.html

Fat arrow funkcije (1)

- U Java Scriptu funkcije mogu biti prosleđene kao parametar prilikom poziva neke druge funkcije

```
setTimeout(function() {
              console.log("setTimeout");
           }, 1000);
```

- Može se izostaviti rezervisana reč `function` i navesti *fat arrow*

```
setTimeout(() => {
              console.log("setTimeout");
           }, 1000);
```

Fat arrow funkcije (2)

- Ukoliko funkcija sadrži samo jedan izraz, mogu se izostaviti vitičaste zagrade i znak ;

```
setTimeout(() => console.log("setTimeout"), 1000);
```

- Jedini parametar možemo navesti bez zagrada - hipotetički

```
setTimeout(s => console.log("setTimeout " + s), 1000);
```

Operatori

- `interval`
  - generiše vrednosti nakon isteka specificiranog intervala u milisekundama
- `subscribe`
  - menja observable u *hot*
  - omogućava prosleđivanje *callback* funkcije koje se izvrši kada se vrednost pojavi **u finalnom toku observable lanca**
- `take`
  - omogućava da ograničimo broj vrednosti koje želimo da uzmemo sa toka
  - prosleđuje se broj željenih vrednosti koje želimo da uzmemo
- `map`
  - uzima izlaz iz toka kao svoj ulaz, konvertuje svaku dobijenu vrednost u neku novu koju šalje je na izlaz

Primeri korišćenja `take` i `map` operatora

```
let obs = interval(1000)
            .pipe(take(3));
obs.subscribe(value => console.log("Subscriber: " + value));
```

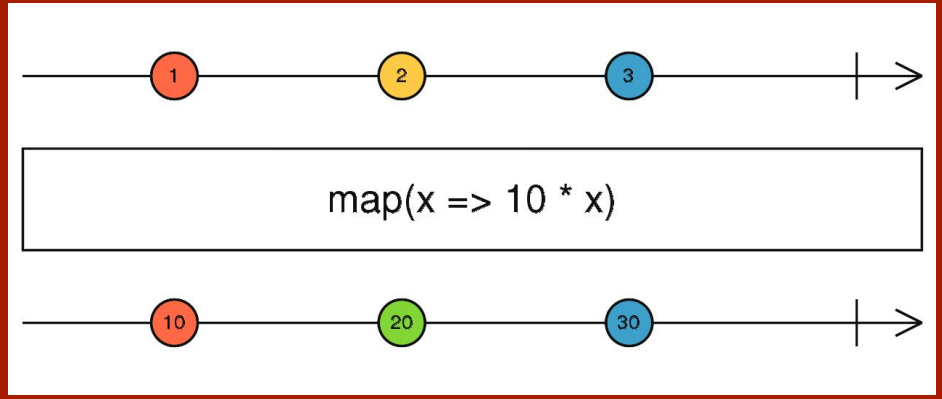
```
Subscriber: 0
Subscriber: 1
Subscriber: 2
```

```
let obs = interval(1000)
           .pipe(take(3), map(v => new Date()));
obs.subscribe(value => console.log("Subscriber: " + value));
```

```
Subscriber: Fri Sep 07 2018 13:17:02 GMT+0200
Subscriber: Fri Sep 07 2018 13:17:03 GMT+0200
Subscriber: Fri Sep 07 2018 13:17:04 GMT+0200
```

Marble dijagrami

- Razumevanje operatora čitanjem tekst je prilično teško
- Rx koristi marble dijagrame da vizuelno opiše funkcionisanje operatora
- Primer dijagrama za map operator:



Korišćenje RxJS u Angular-u

- `EventEmitter` - u pozadini funkcioniše preko `Observable`
- `HttpClient` - HTTP zahtevi se izvršavaju korišćenjem `Observable` (lekcija 9)
- Forme - *Reactive* forme u Angular aplikacijama *expose*-uju `Observable` tok svih ulaznih kontrola (lekcija 7)
- Forme su dobar primer korišćenja reaktivnog programiranja
  - na primeru formi pokazaćemo kako funkcionišu reaktivne forme
  - koristićemo neke koncepte koje ćemo kasnije detaljno obraditi

Primer reaktivne forme

Ukoliko je potrebno, izvršiti nad projektom:
`npm install rxjs@6`

```
import 'rxjs';
…
class FormAppComponent {
  form: FormGroup;
  name = new FormControl("", Validators.required);
  email = new FormControl("", [
    Validators.required,
    Validators.pattern("[^ @]*@[^ @]*")
  ]);
  constructor(fb: FormBuilder) {
    this.form = fb.group({
      "name": this.name,
      "email": this.email
    });
  }
  onSubmit() {
    console.log("Forma potvrdjena!");
  }
}
```

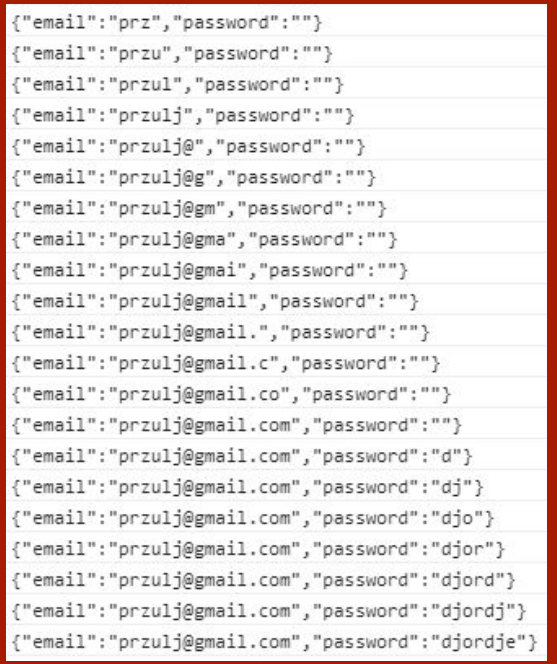
Expose-uje observable

Import modula ReactiveFormModule...

Prijavljivanje na promene u formi

- Instanca `form` ima obeležje `valueChanges` čiji objekat sadrži sve vrednosti polja forme
- Možemo se prijaviti (engl. *subscribe*) i slušati sve promene podataka u formi

```
constructor(fb: FormBuilder) {
  this.form = fb.group({
    "name": this.name,
    "email": this.email
  });
  this.form.valueChanges
    .subscribe(data => console.log(JSON.stringify(data)));
}
```



Procesiranje samo validnih vrednosti

- Kada se radi sa formama, obično želimo da procesiramo samo validne vrednosti unesene u polja
- Ovo se može postići korišćenjem operatora `filter`
- Operator `filter` prihvata funkciju i prosleđuje joj svaku vrednost sa toka
- Ukoliko funkcija vrati `true` filter *publish*-uje ulaznu vrednost na izlazni tok

```
constructor(fb: FormBuilder) {
  this.form = fb.group({
    "name": this.name,
    "email": this.email
  });
  this.form.valueChanges
    .pipe(filter(data => this.form.valid))
    .subscribe(data => console.log(JSON.stringify(data)));
```

```
Angular is running in the development mode. Call
{"email":"przulj@gmail.com","password":"d"}
{"email":"przulj@gmail.com","password":"dj"}
{"email":"przulj@gmail.com","password":"djo"}
{"email":"przulj@gmail.com","password":"djor"}
{"email":"przulj@gmail.com","password":"djord"}
{"email":"przulj@gmail.com","password":"djordj"}
{"email":"przulj@gmail.com","password":"djordje"}
```

Pipes

Namena *pipe*-ova

- Koriste se kada se želi transformisati prikaz podataka u *template*-u
- Koristi se znak `|` (i sam znak se naziva “*pipe*”)
- Primer korišćenja *pipe*-a uz string interpolaciju:

  ```
  {{ 1234.56 | ‘USD’}}
  ```

  će kao rezultat imati

  ```
  USD1,234.56
  ```
- *Pipe*-ovi se mogu ulančavati pa će

  ```
  {{ 1234.56 | ‘USD’ | lowercase }}
  ```

  kao rezultat dati

  ```
  usd1,234.56
  ```

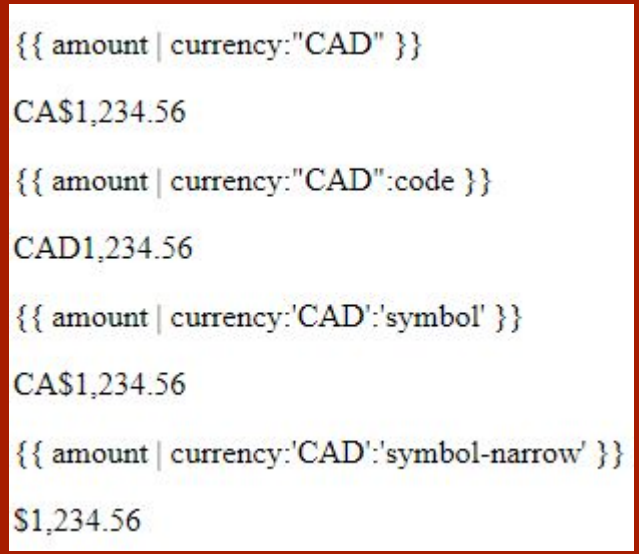
Korišćenje `CurrencyPipe`

- Koristi se za formatiranje valuta
- Prvi argument je skraćenica valute (EUR, USD, CAD…):

  `{{ 1234.56 | currency:”EUR” }}`

  vraća

  `€1,234.56`
- Drugi argument ukazuje da li se želi prikazati
  - kod
  - simbol je jedinstven (default) ili
  - samo simbol valute

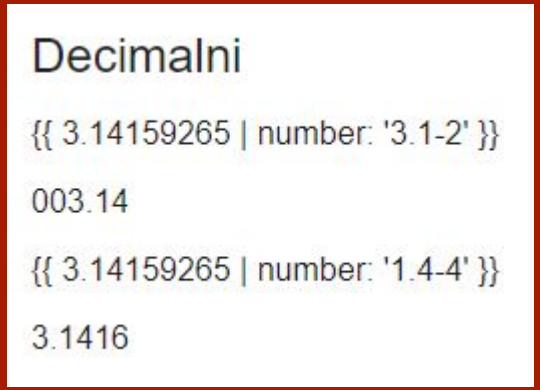


Korišćenje `DatePipe`

```
<mat-card>
  <mat-card-title>Datumi</mat-card-title>
  <p ngNonBindable>{{ dateVal | date: 'shortTime' }}</p>
  <p>{{ dateVal | date: 'shortTime' }}</p>
  <p ngNonBindable>{{ dateVal | date:'fullDate' }}</p>
  <p>{{ dateVal | date: 'fullDate' }}</p>
  <p ngNonBindable>{{ dateVal | date: 'd. M. y.' }}</p>
  <p>{{ dateVal | date: 'd. M. y.' }}</p>
</mat-card>
```

Korišćenje `DecimalPipe`

```
<mat-card>
  <mat-card-title>Decimalni</mat-card-title>
  <p ngNonBindable>{{ 3.14159265 | number: '3.1-2' }}</p>
  <p>{{ 3.14159265 | number: '3.1-2' }}</p>
  <p ngNonBindable>{{ 3.14159265 | number: '1.4-4' }}</p>
  <p>{{ 3.14159265 | number: '1.4-4' }}</p>
</mat-card>
```



Korišćenje `SlicePipe`

```
<mat-card>
  <mat-card-title>Nizovi</mat-card-title>
  <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>
  <p>{{ [1,2,3,4,5,6] | slice:1:3 }}</p>
  <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2 }}</p>
  <p>{{ [1,2,3,4,5,6] | slice:2 }}</p>
  <p ngNonBindable>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>
  <p>{{ [1,2,3,4,5,6] | slice:2:-1 }}</p>
</mat-card>
```

Korišćenje `AsyncPipe`-a

- Dva osnovna načina obrade asinhornih odgovora su:
  - `Promise` i
  - `Observable`
- U oba slučaja, da bismo prikazali odgovor potrebno je da:
  - sačekamo da se izvrši *callback*
  - smestimo rezultat izvršenja *callback* metode u promenljivu
  - prikažemo vrednost promenljive u *template*-u
- `AsyncPipe` prihvata *promise* ili *observable* argument, poziva *then* ili *subscribe* i potom čeka na rezultat

AsyncPipe za Observable

```
export class AsynPipe3Component implements OnDestroy {
  observableData: number;
  subscription: Subscription = null;
  constructor() {
    this.subscribeObservable();
  }
  getObservable() {
    return interval(1000).pipe(take(10), map((v) => v * v));
  }
  subscribeObservable() {
    this.subscription = this.getObservable()
      .subscribe( v => this.observableData = v);
  }
  ngOnDestroy() {
    if (this.subscription)
      this.subscription.unsubscribe();
  }
}
```

```
{{ observableData }}
```

```
export class AsynPipe4Component {
  observable: Observable<number>;

  constructor() {
    this.observable = this.getObservable();
  }

  getObservable() {
    return interval(1000).pipe(take(10),
                               map((v) => v * v));
  }
}
```

```
{{ observable | async }}
```

Efekti primene `AsyncPipe`

- Ne mora eksplicitno da se poziva `subscribe` za `Observable`
- Ne mora u klasi da se smešta vrednost koju dobijamo od `Observable`
- Ne moramo eksplicitno da se odjavljujemo sa `Observable` kada se komponenta uništava
  - sprečava se *memory leak* koji bi se javio ukoliko zaboravimo da se eksplicitno odjavimo

Kreiranje novog *pipe*-a

- Kreirati klasu sa dekoratorom `@Pipe`
- Implementirati metodu `transform`
  - preporučuje se (nije obavezno) implementirati interfejs `PipeTransform`
- Dodati *pipe* klasu u deklaracije modula
- Deklaracije *pipe* klase koja treba da skrati prosleđeni string na 10 znakova:

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({ name: 'shorten' })
export class ShortenPipe implements PipeTransform {
  transform(value: string) {
    return value.substr(0, 10);
  }
}
```

```
{{ title | shorten }}
```

Konfigurisanje *pipe*-a

- Drugi (i svi naredni parametri, ukoliko ih ima) koriste se kao parametri koji mogu da se proslede pri korišćenju *pipe*-a
- Navode se nakon naziva *pipe*-a i dvotačke

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

@Pipe({
  name: 'shorten'
})
export class ShortenPipe implements PipeTransform {
  transform(value: string, limit: number): any {
    if (value.length > limit)
      return value.substr(0, limit);
    else
      return value;
  }
}
```

```
{{ title | shorten : 5 }}
```

Rezime

- Reaktivno programiranje omogućava prijavljivanje na tokove i reagovanje na pojavu nove vrednosti na prijavljenom toku
- Biblioteka RxJS omogućava kreiranje formi i procesiranje popunjenih podataka tek kada su validni
- *Pipe*-ovi omogućavaju transformaciju podataka u željeni oblik prilikom njegove prezentacije
- Kreiranje novog *pipe*-a zahteva definisanje klase sa dekoratorom `@Pipe` i implementiranim `PipeTransform` interfejsom