

Web Frameworks Les 3

Componentcommunicatie, Routing, Reactive Forms en Pipes

Kristof Michiels (team: Sven Mariën, Andie Similon)

Topics

- Communicerende componenten
- Routing in Angular
- Reactive forms
- Pipes

Communicerende componenten

Communicatie tussen componenten

- Heel kort aangeraakt in les 2
- Angular-apps bestaan uit een hiërarchie van componenten
- Om data tussen componenten uit te wisselen gebruiken we:
 - **@Input()** → data van ouder → kind
 - **@Output()** → events van kind → ouder

Data ouder -> kind: @Input()

Ouder (TS + Template)

```
@Component({
  selector: 'app-root',
  template: `<app-gebruiker-card [gebruiker]="actieveGebruiker"></app-gebruiker-card>`
})
export class AppComponent {
  actieveGebruiker = { id: 1, naam: 'Emily' };
}
```

Data ouder -> kind: @Input()

Kind (TS + Template)

```
@Component({
  selector: 'app-gebruiker-card',
  template: `<p>{{ gebruiker.naam }}</p>`
})
export class GebruikerCardComponent {
  @Input() gebruiker!: { id: number; naam: string };
}
```

Event kind -> ouder: @Output()

Kind (TS + Template)

```
@Component({
  selector: 'app-kind',
  template: `<button (click)="stuurBericht()">Klik mij</button>`
})
export class KindComponent {
  @Output() boodschap = new EventEmitter<string>();

  stuurBericht() {
    // Kindcomponent "roept" naar de ouder
    this.boodschap.emit('Hallo vanuit het kind!');
  }
}
```

Event kind -> ouder: @Output()

Ouder (TS + template)

```
@Component({
  selector: 'app-root',
  template: `<app-kind (boodschap)="ontvangBericht($event)"></app-kind>`
})
export class AppComponent {
  ontvangBericht(tekst: string) {
    console.log('Ontvangen van kind:', tekst);
  }
}
```

👉 Duidelijk: **@Output stuurt info van kind → ouder.**

Two-way binding

- Combineert **@Input** (gegeven ontvangen) en **@Output** (wijzigingen terugsturen)
- Gebruik: property `x` + event `xChange` → Angular herkent `[(x)]`

Kind (TS)

```
@Input() beoordeling = 0;
@Output() beoordelingChange = new EventEmitter<number>();

stelBeoordelingIn(nieuweWaarde: number) {
  this.beoordeling = nieuweWaarde;
  this.beoordelingChange.emit(this.beoordeling);
}
```

Two-way binding

Ouder (Template)

```
<app-rating [(beoordeling)]="huidigeBeoordeling"></app-rating>
```

Ouder (TS)

```
export class AppComponent {  
  huidigeBeoordeling = 3;  
}
```

- 👉 Nu kan de ouder zowel **de startwaarde doorgeven** als **de updates automatisch ontvangen**.

Event payloads

- Een `@Output` kan méér doorgeven dan een string of getal
- Vaak wordt een **object** doorgestuurd (bv. geselecteerd product)

Kind (TS)

```
@Output() selecteer = new EventEmitter<{ id: number; naam: string }>();  
  
klik() {  
  this.selecteer.emit({ id: 5, naam: 'Product X' });  
}
```

Event payloads

Ouder (Template)

```
<app-product (selecteer)="toonDetails($event)"></app-product>
```

Ouder (TS)

```
toonDetails(product: { id: number; naam: string }) {  
  console.log('Geselecteerd:', product);  
}
```

ngOnChanges

- Soms moet kind reageren op nieuwe **@Input**-data
- Gebruik **ngOnChanges** om wijzigingen te detecteren

```
export class DetailComponent implements OnChanges {
  @Input() gebruiker!: { id: number; naam: string };

  ngOnChanges(changes: SimpleChanges) {
    if (changes['gebruiker']) {
      console.log('Nieuwe gebruiker ontvangen:', this.gebruiker);
    }
  }
}
```

👉 Lifecycle hook voor logica uit te voeren bij nieuwe input. Binnenkort: moderne Signals!

Routing in Angular 20

Routing in Angular 20

- Angular apps zijn **Single Page Applications (SPA)**
- Routing = tonen van verschillende componenten op basis van URL
- De router beheert:
 - **Routes** (mapping tussen path en component)
 - **Navigatie** (links, knoppen)

Bestanden mbt Routing

1. **app.routes.ts** → definieert de routes
2. **app.config.ts** → configureert de applicatie (providers, incl. router)
3. **main.ts** → start de applicatie op met
`bootstrapApplication`

Routes definieren – app.routes.ts

```
import { Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';
import { UserDetailComponent } from './user-detail.component';
import { AdminComponent } from './admin.component';
import { ProductsComponent } from './products.component';

export const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'user/:id', component: UserDetailComponent },
  { path: 'admin', component: AdminComponent },
  { path: 'products', loadComponent: () =>
    import('./products.component').then(m => m.ProductsComponent) },
];
```

Applicatieconfiguratie – app.config.ts

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes)
  ]
};
```

- 👉 Hier kan je ook andere providers toevoegen, bv.
`provideHttpClient()`

Bootstrapping – main.ts

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app.config';
import { AppComponent } from './app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch(err => console.error(err));
```

👉 `main.ts` start de app op en gebruikt **appConfig** om de router en andere providers te laden.

Navigeren

Template

```
<nav>
  <a routerLink="/">Home</a>
  <a routerLink="/about" routerLinkActive="active">About</a>
</nav>
<router-outlet></router-outlet>
```

- `<router-outlet>` = plaats waar de juiste component verschijnt
- `routerLink` = declaratieve navigatie
- `routerLinkActive="active"` voegt de CSS-kLASSE active toe

URL parameters

- Dynamische routes: /user/:id

Route in app.routes.ts

```
{ path: 'user/:id', component: UserDetailComponent }
```

Uitlezen in component

```
constructor(private route: ActivatedRoute) {  
  this.route.params.subscribe(p => console.log(p['id']));  
}
```

👉 Handig voor detailpagina's (bv. product of profiel)

Query parameters

- Voor filters, zoekopdrachten, paginatie
- URL: /search?term=angular&page=2

Uitlezen in component

```
this.route.queryParams.subscribe(q => {  
  console.log(q['term']);  
});
```

Navigeren met query params

```
this.router.navigate(['/search'], { queryParams: { term: 'angular' } });
```

Vooruitblik: ngOnInit()

- ngOnInit() = lifecycle hook die afgaat zodra de component klaarstaat
- Typisch de plek om opstartlogica te doen, zoals parameters uitlezen

```
export class UserDetailComponent implements OnInit {  
  constructor(private route: ActivatedRoute) {}  
  
  ngOnInit() {  
    this.route.params.subscribe(p =>  
      console.log('User ID:', p['id'])  
    );  
  }  
}
```

Programmatic navigation

- Navigeren vanuit TypeScript i.p.v. template

```
constructor(private router: Router) {}

login() {
  this.router.navigate(['/dashboard']);
}
```

👉 Handig na login, formulieverzending, of events.

Route Guards

- Beperken toegang tot routes
- Typen guards: `canActivate`, `canDeactivate`

Voorbeeld: AuthGuard

```
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  canActivate(): boolean {
    return isLoggedIn();
  }
}
```

Route Guards

Gebruik in route

```
{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard] }
```

👉 Cruciaal voor beveiliging & user flow

Lazy loading

- Grote apps → niet alle code tegelijk laden
- Routes koppelen aan **feature modules/standalone components**

Voorbeeld in `app.routes.ts`

```
{ path: 'products', loadComponent: () =>
  import('./products.component').then(m => m.ProductsComponent) }
```

👉 App start sneller, laadt enkel wat nodig is

Formulieren in Angular 20

Formulieren in Angular

- Formulieren = essentieel voor user input (login, registratie, checkout)
- Angular heeft **2 manieren**:
 1. **Template-driven forms** (met `ngModel`) → eenvoudig maar beperkt
 2. **Reactive forms** (logica in TypeScript) → krachtig, flexibel, testbaar

👉 In moderne Angular-apps gebruiken we **Reactive Forms**

Waarom Reactive Forms?

- **Scheidt logica en HTML** (duidelijker)
- **Meer controle** over waarden en validatie
- **Beter schaalbaar** voor grotere formulieren
- **Makkelijker te testen**
- Past bij moderne Angular-reactiviteit (Signals)

Setup

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

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModuleModule],
  templateUrl: './login.component.html'
})
export class LoginComponent {}
```

👉 Nu kunnen we een FormGroup en controls toevoegen
in de klasse

FormGroup en FormControl

```
import { FormGroup, FormControl, Validators } from '@angular/forms';

form = new FormGroup({
  email: new FormControl('', [Validators.required, Validators.email]),
  password: new FormControl('', Validators.required)
});
```

- **FormControl** = 1 inputveld
- **FormGroup** = verzameling van velden
- **Validators** = regels voor validatie

Template voorbeeld

```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
  <label>Email</label>
  <input type="email" formControlName="email">
  @if (form.controls.email.invalid && form.controls.email.touched) {
    <div>Ongeldig emailadres</div>
  }
  <label>Wachtwoord</label>
  <input type="password" formControlName="password">
  <button type="submit" [disabled]="form.invalid">Login</button>
</form>
```



[formGroup] koppelt TS-model aan HTML

Validators

- **Veelgebruikte ingebouwde validators:**
 - Validators.required
 - Validators.email
 - Validators.minLength(n)
 - Validators.maxLength(n)
 - Validators.pattern(regex)

Zelfgemaakte validator (voorbeeld)

```
function minVijf(control: FormControl) {
  return control.value && control.value.length >= 5 ? null : { minVijf: true };
}
```

Waarden lezen & updaten

```
onSubmit() {
  console.log(this.form.value); // { email: '...', password: '...' }
  console.log(this.form.controls.email.value);
}

// Updaten
this.form.patchValue({ email: 'test@example.com' });
```

- `patchValue` → update enkel opgegeven velden
- `setValue` → vereist ALLE velden

FormArray (dynamische velden)

```
import { FormArray, FormControl } from '@angular/forms';
phones = new FormArray([ new FormControl('') ]);
```

```
<div formArrayName="phones">
  @for (ctrl of phones.controls; let i = $index; track $index) {
    <input [formControlName]="i">
  }
</div>
```

👉 Handig voor lijsten zoals telefoonnummers of tags

Foutmeldingen tonen

```
<input formControlName="email">
@if (form.controls.email.touched && form.controls.email.hasError('required')) {
  <div>Email is verplicht</div>
}
@if (form.controls.email.touched && form.controls.email.hasError('email')) {
  <div>Ongeldig emailadres</div>
}
```

👉 Toon foutmeldingen pas **na interactie** (touched/dirty)

Best practices

- Hou validatie in TypeScript, niet in HTML
- Gebruik duidelijke en korte foutmeldingen
- Structureer grotere formulieren met groepen of FormArray
- Test je validators apart
- Disable submit-knop bij ongeldig formulier

Pipes

Pipes

- Pipes transformeren waarden **in de template** (presentatielogica)
- Declaratief: {{ value | pipeName:arg1:arg2 }}
- Pipes kun je **chainen**: {{ price | number:'1.2-2' | currency:'EUR' }}
- In Angular 20 zijn pipes vaak **standalone** → importeer ze in je component

Standalone pipes importeren

```
@Component({
  selector: 'app-order',
  standalone: true,
  imports: [CommonModule, CapitalizePipe, EuroPipe],
  template: `
    <h2>{{ title | capitalize }}</h2>
    <p>{{ total | euro }}</p>
  `

})
export class OrderComponent {
  title = 'overzicht';
  total = 1999.9;
}
```

Locale & Europese context

Stel Europese notatie in (komma voor decimalen, €-symbool, datumstijl):

```
import { bootstrapApplication, LOCALE_ID, DEFAULT_CURRENCY_CODE } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeNlBE from '@angular/common/locales/nl-BE';

registerLocaleData(localeNlBE);

bootstrapApplication(AppComponent, {
  providers: [
    { provide: LOCALE_ID, useValue: 'nl-BE' },
    { provide: DEFAULT_CURRENCY_CODE, useValue: 'EUR' }
  ]
});
```

Daarna passen `date`, `number`, `percent`, `currency` automatisch EU-notatie toe.

Vaak gebruikte pipes: date

```
<p>Vandaag: {{ today | date:'fullDate' }}</p>      <!-- dinsdag 14 oktober 2025 -->
<p>Kort: {{ today | date:'dd/MM/yyyy' }}</p>       <!-- 14/10/2025 -->
<p>Tijd: {{ now | date:'HH:mm' }}</p>            <!-- 09:30 -->
<p>Tijdzone: {{ now | date:'dd/MM/yyyy, HH:mm zzzz' }}</p>
```

- Tip: gebruik duidelijke, consistentie patronen in je toepassingen

Vaak gebruikte pipes: currency

```
<p>Prijs: {{ 12.5 | currency:'EUR':'symbol' }}</p>           <!-- € 12,50 -->
<p>Met code: {{ 12.5 | currency:'EUR':'code' }}</p>          <!-- EUR 12,50 -->
<p>Precisie: {{ 1999.9 | currency:'EUR':'symbol':'1.2-2' }}</p> <!-- € 1.999,90 -->
```

“ Met `DEFAULT_CURRENCY_CODE: 'EUR'` kun je `currency` zonder code gebruiken. ”

Andere nuttige pipes

- `number` → Europese getalnotatie (`1.234,56`)
- `percent` → percentages (`12,3 %`)
- `titlecase` / `uppercase` / `lowercase` → eenvoudige tekstopmaak

Wanneer custom pipes maken?

- WEL, als je telkens dezelfde opmaak of transformatie nodig hebt in de UI
(bv. hoofdletter zetten, bedrag tonen in €)
- WEL, om herhaling in je templates te vermijden
- NIET voor business-logica of rekenregels → dat hoort in een service (zie later)
- NIET voor async werk (zoals data ophalen) of dingen met side-effects → hoort in component/service
- NIET voor zware berekeningen die je app traag maken

Voorbeeld: **capitalize**

```
@Pipe({ name: 'capitalize', standalone: true, pure: true })
export class CapitalizePipe implements PipeTransform {
  transform(value: string | null | undefined): string {
    if (!value) return '';
    return value.charAt(0).toUpperCase() + value.slice(1);
  }
}
```

```
<p>{{ 'antwerpen' | capitalize }}</p> <!-- Antwerpen -->
```

Voorbeeld: ibanMask (IBAN leesbaar maken)

```
@Pipe({ name: 'ibanMask', standalone: true, pure: true })
export class IbanMaskPipe implements PipeTransform {
  transform(iban: string | null | undefined): string {
    if (!iban) return '';
    return iban.replace(/\s+/g, '').replace(/(\.{4})/g, '$1 ').trim();
  }
}
```

```
<p>IBAN: {{ 'BE71096123456769' | ibanMask }}</p> <!-- BE71 0961 2345 6769 -->
```

Voorbeeld: vat (BTW-nummers formatteren)

```
@Pipe({ name: 'vat', standalone: true, pure: true })
export class VatPipe implements PipeTransform {
  transform(vat: string | null | undefined): string {
    if (!vat) return '';
    const clean = vat.replace(/[^A-Za-z0-9]/g, '').toUpperCase();
    if (clean.startsWith('BE')) {
      const digits = clean.slice(2).padStart(10, '0');
      return `BE${digits.slice(0,4)}.${digits.slice(4,7)}. ${digits.slice(7)}`;
    }
    return clean; // fallback
  }
}
```

```
<p>BTW: {{ 'be123456789' | vat }}</p> <!-- BE0123.456.789 -->
```

Voorbeeld: euro (consistent €-bedrag)

```
@Pipe({ name: 'euro', standalone: true, pure: true })
export class EuroPipe implements PipeTransform {
  transform(value: number | null | undefined, min = 2, max = 2): string {
    const n = value ?? 0;
    return new Intl.NumberFormat('nl-BE', {
      style: 'currency', currency: 'EUR',
      minimumFractionDigits: min, maximumFractionDigits: max
    }).format(n);
  }
}
```

```
<p>{{ 1999.9 | euro }}</p> <!-- € 1.999,90 -->
```

Bedankt voor je aandacht! 😊

Vragen? Feedback?

kristof.michiels01@ap.be