Angular - cheat sheet

Merk op: deze samenvatting is geen vervanger voor het officiële lesmateriaal. Het zijn eigen uitgewerkte nota's bij de videolessen. Bedoeling is aan te geven hoe belangrijk het is om de aangereikte leerstof op je eigen manier actief te verwerken, bvb door het maken van een eigen samenvatting. Zo behoud je niet alleen het overzicht, maar kom je ook goed voorbereid aan de start van elk labo.

Inhoud

- Modules
- Routing
- Services

Basics

Een nieuw Angular-project aanmaken

Installatie Angular CLI:

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

Maak een nieuw project door onderstaand commando uit te voeren. De CLI zal een aantal vragen stellen over de functies die je wilt opnemen in je project, zoals routing en de stijlsheetindeling. Navigeer vervolgens naar de projectfolder en start de lokale ontwikkelserver met ng serve. De Angular CLI zal je applicatie bouwen en een lokale ontwikkelserver starten. Je kunt je nieuwe Angular-applicatie zien door in een browser naar `http://localhost:4200/` te gaan.

```
ng new mijn-nieuw-project
cd mijn-nieuw-project
ng serve
```

Package.json

Het `package.json` bestand in een Angular (of enige andere Node.js) applicatie speelt een cruciale rol in het beheer van de afhankelijkheden (packages) van de toepassing. Het bevat metadata over het project en een lijst van afhankelijkheden die nodig zijn om het project uit te voeren en te ontwikkelen.

- **Project Metadata:** Bevat basisinformatie zoals de naam, versie, auteur, en beschrijving van het project.
- **Dependencies en DevDependencies:** Lijst de packages op die nodig zijn om je applicatie te draaien (`dependencies`) en om het te ontwikkelen en testen (`devDependencies`).
- **Scripts:** Definieert scripts die kunnen worden uitgevoerd, zoals `start`, `build`, en `test`.

Packages Toevoegen/Verwijderen:

1. **Handmatig Toevoegen:**
   Om een package handmatig toe te voegen, open je `package.json` en voeg je de package toe aan de `dependencies` of `devDependencies` sectie met de gewenste versie. Bijvoorbeeld:

   ```
   "dependencies": {
     "some-package": "^1.0.0"
   }
   ```

   Na het toevoegen, voer je `npm install` uit in de terminal om de package daadwerkelijk te installeren.

2. **Automatisch Toevoegen:**
   Je kunt ook `npm install some-package --save` gebruiken om een package toe te voegen. Dit voegt de package toe aan je `dependencies` en installeert het tegelijkertijd. Voor dev dependencies, gebruik `npm install some-package --save-dev`.

3. **Packages Verwijderen:**
   Om een package te verwijderen, verwijder je eenvoudigweg de regel uit `package.json` en voer je `npm install` uit, of gebruik `npm uninstall some-package`.

Aandachtspunten:

- Het is belangrijk om de juiste versienotatie te gebruiken bij het toevoegen van packages. De caret (`^`) en tilde (`~`) symbolen hebben specifieke betekenissen in termen van welke versies zijn toegestaan.
- Na het wijzigen van `package.json` is het essentieel om `npm install` uit te voeren om ervoor te zorgen dat je `node_modules` map en je `package-lock.json` bestand up-to-date zijn met je wijzigingen.

Snel wegwijs in je Angular toepassing

Een met Angular CLI aangemaakte Angular-toepassing heeft een gestandaardiseerde mappenstructuur en enkele belangrijke bestanden. Deze bestanden en mappen vormen de basisstructuur van een Angular-toepassing. Het Angular CLI biedt een consistente structuur die het makkelijk maakt om nieuwe componenten, services, en andere bestanden toe te voegen en te beheren. Dit helpt bij het organiseren van de applicatie en maakt het makkelijker voor andere ontwikkelaars om het project te begrijpen en eraan bij te dragen.

1. `/src` **Map:**

- `app/`: Bevat de componenten, services, en andere Angular-specifieke bestanden van de applicatie.
  - `app.component.*`: De root component van je applicatie.
  - `app.module.ts`: De root module die componenten, services, en andere modules declareert.
- `assets/`: Voor statische bestanden zoals afbeeldingen, fonts, etc.
- `environments/`: Bevat configuratiebestanden voor verschillende omgevingen (bijv. productie, ontwikkeling).
- `index.html`: De hoofdpagina van de applicatie.
- `main.ts`: Het startpunt van de applicatie, waar de root module wordt opgestart.
- `styles.css` (of `.scss`): Globale stijlen voor de gehele applicatie.

2. **Root Map:**
   - `package.json`: Bevat zoals gezegd metadata en lijst van afhankelijkheden van het project.
   - `angular.json`: Angular CLI configuratiebestand waarin opties voor het Angular project zijn gespecificeerd, zoals build- en serve-opties.
   - `tsconfig.json`: Bevat de TypeScript compiler opties.
   - `node_modules/`: Bevat alle geïnstalleerde Node.js modules die zijn gespecificeerd in `package.json`.
3. **Testbestanden:**
   - `.spec.ts`: Testbestanden geschreven in Jasmine en gerund door Karma, gelegen naast de bestanden die ze testen.

Componenten

Wat verstaan we onder componenten?

Angular-toepassingen zijn gebouwd rond het concept van componenten, die de bouwstenen van de applicatie vormen. Hier is een kort en bondig overzicht van de opbouw met betrekking tot componenten:

1. **Componenten Structuur:**
   - Elke component in Angular bestaat in principe uit drie kernbestanden:

    - **HTML Template Bestand:** Definieert de view of de gebruikersinterface van de component (bijv. `app.component.html`).
    - **Class Bestand:** Een TypeScript klasse waarin de logica van de component is gedefinieerd (bijv. `app.component.ts`). Deze klasse is gemarkeerd met de `@Component` decorator, die metadata zoals de template, stijlen en selector specificeert.
    - **CSS/SCSS Bestand:** Bevat de stijlen die specifiek zijn voor de component (bijv. `app.component.css` of `.scss`).
2. **Component Decorator:**
    - De `@Component` decorator identificeert de klasse direct eronder als een Angular component en geeft metadata door die bepaalt hoe de component moet worden verwerkt, geïnstantieerd en gebruikt.
3. **Data Binding:**
    - Componenten gebruiken data binding om te communiceren met hun templates. Dit kan zijn via property binding, event binding, of two-way binding (zie verder).
4. **Input en Output:**
    - Componenten kunnen data ontvangen van hun oudercomponenten via `@Input()` en gebeurtenissen naar oudercomponenten sturen via `@Output()`.
5. **Levenscyclus Hooks:**
    - Angular biedt levenscyclus hooks zoals `ngOnInit`, `ngOnChanges`, `ngOnDestroy`, enz., waarmee je specifieke code kunt uitvoeren op bepaalde momenten in het leven van de component.
6. **Nesting en Herbruikbaarheid:**
    - Componenten kunnen worden genest binnen andere componenten, wat hergebruik van code en functionaliteit bevordert. Dit maakt de Angular-toepassing modulair en gemakkelijker te onderhouden.

Deze component-gebaseerde architectuur maakt Angular-applicaties zeer modulair, waardoor ze makkelijk zijn uit te breiden en te onderhouden. Componenten kunnen onafhankelijk worden ontwikkeld, getest en hergebruikt, wat bijdraagt aan de efficiëntie en effectiviteit van de ontwikkeling.

In een Angular-toepassing speelt de `AppComponent` de rol van de root-component. Deze component dient als het startpunt van de applicatie en de ouder van alle

andere componenten. De `AppComponent` wordt als eerste geladen en vormt de basis van de applicatie-view hierarchie. Alle andere componenten worden hierbinnen of hieronder in de componentenboom geplaatst. Het is in feite het skelet waarop de hele applicatie is opgebouwd, en het fungeert als de container voor de rest van de applicatie.

Een nieuwe component aanmaken

Om een nieuwe (child-)component in een Angular-toepassing aan te maken, kun je zowel de Angular CLI als een handmatige methode gebruiken:

1. **Met Angular CLI:**
   - Open je terminal of command prompt.
   - Navigeer naar de root-directory van je Angular-project.
   - Gebruik het volgende commando:

     ```
     ng generate component [component-naam]
     ```

   - Vervang `[component-naam]` met de gewenste naam van je component. Angular CLI creëert automatisch een nieuwe map met alle relevante bestanden (TypeScript, HTML, CSS) binnen de `src/app` directory van je project.
2. **Handmatig:**
   - Ga naar de `src/app` directory van je project.
   - Maak handmatig een nieuwe map aan voor je component.
   - Binnen deze map, maak je de volgende bestanden:
     - `[component-naam].component.ts` voor de TypeScript klasse.
     - `[component-naam].component.html` voor de template.
     - `[component-naam].component.css` (of `.scss`) voor de stijlen.
   - Definieer de componentklasse in het `.ts` bestand en decoreer het met `@Component`, inclusief de `selector`, `templateUrl`, en `styleUrls`.

De Angular CLI methode is sneller en wordt aanbevolen, omdat het automatisch de nodige bestanden en boilerplate code genereert, terwijl de handmatige methode meer controle biedt over de structuur en organisatie van je componenten.

In een Angular-project is het gebruikelijk om componenten te organiseren op een manier die de structuur en hiërarchie van de applicatie weerspiegelt. Hoewel het

technisch mogelijk is om alle componenten direct in de `src/app` directory aan te maken, verdient het onderbrengen van child-components in de directory van hun parent-component de voorkeur.

In de praktijk zou je bijvoorbeeld een directorystructuur kunnen hebben waarin `ParentComponent` zijn eigen map heeft, en alle bijbehorende `ChildComponents` worden binnen deze map geplaatst. Dit helpt om de relatie tussen de componenten visueel en organisatorisch duidelijk te maken.

Interpolation

Interpolatie in Angular is een techniek om data vanuit de component (de TypeScript code) te binden aan de HTML-view. Het wordt gebruikt om dynamische waarden in de HTML te tonen. Deze waarden zijn meestal eigenschappen van de component.

Een basisvoorbeeld: stel, je hebt een component met een eigenschap `naam` in TypeScript. Je wilt deze naam in de HTML-view weergeven. Dit doe je door in het HTML-bestand de eigenschap tussen dubbele accolades te plaatsen, zoals `{{ naam }}`.

TypeScript (component.ts):

```
export class MijnComponent {
  naam = 'Kristof';
}
```

HTML (component.html):

```
<p>Hallo, mijn naam is {{ naam }}!</p>
```

In dit voorbeeld zal in de browser "Hallo, mijn naam is Jan!" getoond worden. Wanneer de waarde van `naam` in de component verandert, zal deze wijziging automatisch in de view weerspiegeld worden dankzij Angular's data-binding mechanisme.

Pipes

In Angular zijn pipes functies die je in templates gebruikt om waarden te transformeren voordat ze worden weergegeven. Ze bieden een handige manier om data te formatteren zonder de oorspronkelijke data te wijzigen. Pipes kunnen worden onderverdeeld in twee categorieën: built-in pipes en custom pipes.

Built-in Pipes:
Dit zijn de pipes die standaard worden meegeleverd met Angular. Enkele van de meest gebruikte zijn:

- **DatePipe:** Transformeert een datumwaarde naar een leesbare string.
- **UpperCasePipe en LowerCasePipe:** Converteert tekst naar respectievelijk alle hoofdletters of kleine letters.
- **DecimalPipe:** Formatteert een getal als tekst met een specifiek aantal decimalen.
- **CurrencyPipe:** Transformeert een getal naar een valuta-string.
- **PercentPipe:** Transformeert een getal naar een percentage-string.

Voorbeeld van Built-in Pipe:

```
<p>Today is {{ today | date:'fullDate' }}</p>
```

In dit voorbeeld wordt de `DatePipe` gebruikt om de huidige datum (`today`) te formatteren als een volledige datumstring.

Custom Pipes:
Als de built-in pipes niet voldoen aan je vereisten, kun je je eigen custom pipes maken.

Voorbeeld van het Aanmaken van een Custom Pipe:

1. **Definieer de Pipe:**

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

   @Pipe({
     name: 'exclamation'
   })
   export class ExclamationPipe implements PipeTransform {
     transform(value: string, times: number = 1): string {
       return value + '!'.repeat(times);
     }
   }
   ```

   Deze custom pipe voegt een aantal uitroeptekens toe aan het einde van een string. Het aantal uitroeptekens wordt bepaald door de parameter `times`.

2. **Gebruik de Custom Pipe in een Template:**

   ```
   <p>{{ 'Hallo' | exclamation:3 }}</p>
   ```

   Dit zou renderen als `Hallo!!!` in de browser.

Door zowel built-in als custom pipes te gebruiken, kun je een hoge mate van flexibiliteit en aanpassing bereiken in de manier waarop data wordt gepresenteerd in een Angular-applicatie, zonder de oorspronkelijke data of logica van je componenten te wijzigen.

Directives

In Angular zijn directives instructies die je aan HTML-elementen geeft om hun gedrag en/of uiterlijk aan te passen. Directives zijn krachtige hulpmiddelen die het mogelijk maken om interactieve en dynamische webapplicaties te bouwen. Twee van de meest gebruikte built-in directives zijn `ngIf` en `ngFor`.

1. **ngIf Directive:** `ngIf` is een structural directive die wordt gebruikt om delen van de DOM conditioneel te renderen. Als de expressie die aan `ngIf` wordt doorgegeven `true` evalueert, wordt het element getoond; als het `false` evalueert, wordt het element verwijderd uit de DOM.

   Voorbeeld van ngIf:

   ```
   <p *ngIf="showMessage">Dit is een belangrijk bericht!</p>
   ```

   In dit voorbeeld zal de `<p>` tag alleen worden getoond als `showMessage` `true` is.

2. **ngFor Directive:** `ngFor` is een andere structural directive die wordt gebruikt om een set van HTML-elementen te renderen voor elk item in een array.

   Voorbeeld van ngFor:

   ```
   <ul>
     <li *ngFor="let item of items">{{ item }}</li>
   </ul>
   ```

   Hier zal voor elk item in de array `items` een `<li>` element worden gecreëerd. Als `items` bijvoorbeeld de array `['Appel', 'Banaan', 'Citroen']` bevat, zal dit resulteren in drie `<li>` elementen met de tekst 'Appel', 'Banaan', en 'Citroen'.

Deze directives verrijken de HTML-template van een Angular-component door het toevoegen van logica en conditionele structuur, wat resulteert in een dynamischere en interactievere gebruikersinterface.

Ter info: Angular bevat een aantal ingebouwde directives, waaronder:

1. **Structural Directives:**
   - `ngIf`
   - `ngFor`
   - `ngSwitch`
   - `ngSwitchCase`
   - `ngSwitchDefault`
   - `ngTemplateOutlet`
2. **Attribute Directives:**
   - `ngClass`
   - `ngStyle`
   - `ngModel`

Deze directives zijn essentieel voor het controleren van de rendering en het gedrag van HTML-elementen in Angular templates.

Property binding

Property binding in Angular is een vorm van one-way data binding waarbij waarden van de component-klasse worden doorgegeven aan eigenschappen van HTML-elementen of child-componenten in de template. Dit wordt gedaan met behulp van vierkante haakjes `[]` rond de eigenschapnaam.

Voorbeelden van Property Binding:

1. **Binding naar HTML Element Eigenschappen:**
   Stel, je hebt een eigenschap `imageUrl` in je component die een URL van een afbeelding bevat. Je kunt deze waarde binden aan de `src` eigenschap van een `<img>` tag in je template:

   ```
   // In je component
   imageUrl = 'pad/naar/afbeelding.jpg';
   ```

```
<!-- In je template -->
<img [src]="imageUrl">
```

Hier wordt de waarde van `imageUrl` uit de component gebonden aan de `src` eigenschap van de `<img>` tag.

2. **Binding naar Component Eigenschappen:**
   Als je een child-component hebt met een `@Input()` eigenschap, kun je die binden aan een waarde van de parent-component. Stel dat `ChildComponent` een input eigenschap `childTitle` heeft:

   ```
   // In ChildComponent
   @Input() childTitle: string;
   ```

   In de parent-component kun je deze binden als volgt:

   ```
   <!-- In de template van de parent-component -->
   <app-child [childTitle]="parentTitle"></app-child>
   ```

   Hier wordt de waarde van `parentTitle` uit de parent-component gebonden aan de `childTitle` eigenschap van `ChildComponent`.

3. **Binding naar Directive Eigenschappen:**
   Angular directives zoals `ngStyle` en `ngClass` kunnen ook gebruik maken van property binding. Bijvoorbeeld, je kunt de stijlen dynamisch aanpassen met `ngStyle`:

   ```
   // In je component
   currentStyles = {color: 'blue', 'font-size': '12px'};
   ```

   ```
   <!-- In je template -->
   <div [ngStyle]="currentStyles">Tekst met dynamische stijle
   ```

   Hier worden de stijlen gedefinieerd in `currentStyles` toegepast op de `<div>`.

Property binding is een krachtige manier in Angular om de applicatie dynamisch en interactief te maken, waarbij de dataflow altijd van de component-klasse naar de template gaat. Dit zorgt voor een duidelijke scheiding tussen de logica van de applicatie en de UI.

Property binding vs interpolation

Beiden worden gebruikt voor het doorgeven van data van de component naar de template, maar ze verschillen in hun gebruik en functionaliteit:

1. **Gebruik en Syntax:**
   - **Property Binding:** Gebruikt vierkante haakjes `[]` om een eigenschap van een HTML-element of een directive te binden aan een expressie. Bijvoorbeeld: `[src]="imageUrl"` bindt de `src` eigenschap van een `<img>` element aan de `imageUrl` variabele in de component.
   - **Interpolatie:** Gebruikt dubbele accolades `{{ }}` om een string representatie van een expressie in te voegen in de tekstcontent of attributen van HTML-elementen. Bijvoorbeeld: `src="{{imageUrl}}"` voegt de waarde van `imageUrl` in als deel van de `src` attribuut als string.
2. **Wanneer te Gebruiken:**
   - **Property Binding:** Wordt gebruikt wanneer je een waarde wilt binden aan een eigenschap van een element of component. Dit is nuttig voor dynamische waarden en complexe expressies, en is essentieel wanneer je met niet-string waarden werkt.
   - **Interpolatie:** Is beperkt tot string waarden of waarden die als string kunnen worden gerepresenteerd. Het is handig voor het invoegen van tekstuele data in de template.
3. **Type Binding:**
   - **Property Binding:** Kan elk type data binden dat door de eigenschap wordt ondersteund, inclusief strings, booleans, objecten, arrays, etc.
   - **Interpolatie:** Alles wat binnen de accolades staat wordt uiteindelijk omgezet in een string. Dit maakt het minder flexibel voor het binden van niet-string waarden.

Voorbeeld Vergelijking:

- Met property binding (bindt een boolean aan een button's `disabled` eigenschap):

  ```
  <button [disabled]="isButtonDisabled">Klik Mij</button>
  ```

  Hier, als `isButtonDisabled` `true` is, wordt de knop gedeactiveerd.
- Met interpolatie (voegt tekst in binnen een element):

```
<div>Mijn afbeelding URL is: {{ imageUrl }}</div>
```

Dit zet de waarde van `imageUrl` om naar een string en voegt deze in in de `<div>`.

Samengevat, property binding is krachtiger en veelzijdiger, vooral voor het binden van non-string waarden, terwijl interpolatie voornamelijk wordt gebruikt voor het weergeven van tekstuele data.

Event binding

Event binding in Angular is een mechanisme waarmee je kunt reageren op gebruikersinteracties zoals klikken, toetsaanslagen, muisbewegingen, enz. Het stelt een Angular-component in staat om te luisteren naar dergelijke gebeurtenissen op HTML-elementen in de template en vervolgens een specifieke methode of actie uit te voeren als reactie op die gebeurtenissen.

De syntax voor event binding in Angular is eenvoudig: je gebruikt ronde haakjes `()` rondom de eventnaam en wijst deze toe aan een methodenaam van de component-klasse.

Voorbeelden van Event Binding:

1. **Reageren op Klikgebeurtenissen:**
   Stel je hebt een methode `onClickMe()` in je componentklasse die je wilt uitvoeren wanneer een gebruiker op een knop klikt.

   ```
   // In je component klasse
   export class AppComponent {
     onClickMe() {
       console.log('De knop is geklikt!');
     }
   }
   ```

   ```
   <!-- In je template -->
   <button (click)="onClickMe()">Klik op mij</button>
   ```

   Hier, wanneer de knop wordt geklikt, zal de `onClickMe()` methode worden uitgevoerd.

2. **Luisteren naar Toetsaanslagen:**
   Je kunt ook luisteren naar toetsaanslagen op bijvoorbeeld een inputveld.

```
// In je component klasse
onKey(event: any) {
  console.log('Toetsaanslag:', event.target.value);
}
```

```
<!-- In je template -->
<input (keyup)="onKey($event)">
```

   Elke keer wanneer een toets wordt losgelaten terwijl de focus op het inputveld ligt, wordt `onKey()` aangeroepen en logt het de huidige waarde van het veld.

3. **Andere Evenementen:**
   Je kunt luisteren naar een verscheidenheid aan andere DOM-evenementen, zoals
   `mouseover`, `mouseout`, `keydown`, `blur`, etc.

```
// In je component klasse
export class AppComponent {
  onMouseOver() {
    console.log('Muis over het element!');
  }
}
```

```
<!-- In je template -->
<div (mouseover)="onMouseOver()">Beweeg je muis over deze tek
```

Template reference variables

Template reference variables in Angular bieden een manier om te verwijzen naar een DOM-element of een Angular-component binnen de template. Dit stelt je in staat om direct toegang te krijgen tot de eigenschappen en methoden van dat element of die component. Ze worden gedefinieerd met een voorafgaande hash (`#`) in de template.

Hoe gebruik je Template Reference Variables:

1. **Definiëren in de Template:**
   Je definieert een template reference variable door een naam toe te wijzen met een hash. Bijvoorbeeld:
   `<input #myInput>` maakt een referentie naar het `input` element met de naam `myInput`.

2. **Referentie in de Template:**
   Je kunt deze referentie vervolgens gebruiken binnen de template om toegang te krijgen tot eigenschappen of methoden van het element. Bijvoorbeeld, je kunt een knop gebruiken om de waarde van het inputveld te loggen zonder een binding naar de componentklasse te gebruiken.

Voorbeeld:

Stel, je wilt de waarde van een inputveld loggen wanneer een gebruiker op een knop klikt:

```
<!-- In je Angular template -->
<input #myInput type="text">
<button (click)="logInputValue(myInput.value)">Log waarde</bu
```

In je componentklasse:

```
// In je component klasse
export class AppComponent {
  logInputValue(value: string) {
    console.log('Inputwaarde:', value);
  }
}
```

In dit voorbeeld:

- `#myInput` is een template reference variable die verwijst naar het `input` element.
- Wanneer de knop wordt geklikt, roept de `(click)` event handler de `logInputValue()` methode aan met `myInput.value` als parameter. Dit stuurt de huidige waarde van het inputveld naar de methode in de componentklasse.
- De `logInputValue()` methode logt vervolgens de waarde naar de console.

Deze aanpak is bijzonder handig voor het direct benaderen van DOM-elementen of het werken met Angular-componenten binnen de template, zonder dat je extra logica in de TypeScript-klasse nodig hebt.

Two-way binding

Two-way binding in Angular is een mechanisme dat het mogelijk maakt om een model (in de component-klasse) en een view (in de template) te synchroniseren, zodat wijzigingen in het ene automatisch worden weerspiegeld in het andere, en omgekeerd. Dit wordt bereikt met behulp van de `ngModel` directive, een combinatie van zowel property binding (van de component naar de view) als event binding (van de view terug naar de component).

Hoe Two-Way Binding Werkt:

In de praktijk wordt two-way binding vaak gebruikt in formulierelementen zoals `input`, `select`, en `textarea`. Je gebruikt de `[(ngModel)]` syntax om de waarde van een formulierelement te binden aan een eigenschap in je componentklasse.

Voorbeeld van Two-Way Binding:

Stel, je hebt een eenvoudige `input` veld in je template en je wilt de waarde ervan binden aan een eigenschap `textInput` in je componentklasse.

1. **Component Klasse:**
   Hier definieer je een eigenschap `textInput`.

   ```
   export class AppComponent {
     textInput = '';
   }
   ```

2. **Template:**
   In de template gebruik je de `[(ngModel)]` syntax om een two-way binding te creëren tussen het `input` veld en de `textInput` eigenschap.

   ```
   <input [(ngModel)]="textInput" type="text">
   <p>De ingevoerde tekst is: {{ textInput }}</p>
   ```

In dit voorbeeld:

- Wanneer je iets typt in het `input` veld, wordt de `textInput` eigenschap in de componentklasse automatisch bijgewerkt met die waarde.
- Tegelijkertijd, als de waarde van `textInput` programmatisch wordt gewijzigd in de componentklasse, zal de weergave in het `input` veld automatisch worden

bijgewerkt.

Deze vorm van data binding is bijzonder nuttig in formulieren en interactieve interfaces, waar een naadloze synchronisatie tussen de gebruikersinterface en de onderliggende data vereist is.

Parent-child gegevensuitwisseling

Parent-child gegevensuitwisseling in Angular verwijst naar de communicatie tussen een parent component en een child component. Dit wordt typisch gerealiseerd door middel van `@Input()` en `@Output()` decorators.

@Input() - Data van Parent naar Child:
De
`@Input()` decorator in de child component stelt het in staat om data te ontvangen van de parent component.

@Output() en EventEmitter - Data van Child naar Parent:
De
`@Output()` decorator, in combinatie met `EventEmitter`, wordt gebruikt in de child component om gebeurtenissen (of data) naar de parent component te sturen.

Voorbeeld:

Stel, je hebt een parent component `AppComponent` en een child component `ChildComponent`.

1. **Child Component met `@Input()` en `@Output()`:**

```
import { Component, Input, Output, EventEmitter } from '@a

@Component({
  selector: 'app-child',
  template: `<p>{{ receivedData }}</p>
             <button (click)="sendDataToParent()">Stuur na
})
export class ChildComponent {
  @Input() receivedData: string; // Data van parent
  @Output() dataEvent = new EventEmitter<string>(); // Stu

  sendDataToParent() {
    this.dataEvent.emit('Data van Child');
```

```
  }
}
```

2. **Parent Component die Data Verstuurt en Ontvangt:**

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

@Component({
  selector: 'app-root',
  template: `<app-child [receivedData]="parentData"
             (dataEvent)="receiveData($event)"></app-chil
})
export class AppComponent {
  parentData = 'Data van Parent';
  receivedChildData: string;

  receiveData(data: string) {
    this.receivedChildData = data;
  }
}
```

In dit voorbeeld:

- De `AppComponent` (parent) stuurt data naar `ChildComponent` via de `@Input()` eigenschap `receivedData`.
- Wanneer de gebruiker op de knop in `ChildComponent` klikt, stuurt het component een bericht terug naar `AppComponent` met behulp van `EventEmitter` via de `@Output()` eigenschap `dataEvent`.
- `AppComponent` vangt dit op in de `receiveData` methode en verwerkt de data.

Deze methode van gegevensuitwisseling zorgt voor een duidelijk gedefinieerde interface tussen parent en child components en bevordert de modulariteit en herbruikbaarheid van componenten.

Ook vanuit de child kunnen gegevens worden teruggestuurd naar de parent. Mbv @Output decorator en een event mechanisme. Buiten de scope van de cursus.

Modules

In Angular zijn modules een cruciaal organisatorisch concept dat helpt bij het groeperen van gerelateerde delen van een applicatie. Een Angular-module, vaak aangeduid met `@NgModule`, is een klasse met een `@NgModule()` decorator die functies en afhankelijkheden van de module definieert.

De hoofddoelen van een Angular-module zijn:

1. **Organisatie**: Modules bundelen gerelateerde code samen, zoals componenten, services, directives en pipes. Dit bevordert een schone, overzichtelijke structuur in de applicatie.
2. **Encapsulatie**: Door onderdelen in modules te plaatsen, kan de zichtbaarheid van deze onderdelen beperkt worden. Dit betekent dat componenten en services binnen een module normaal gesproken enkel beschikbaar zijn voor andere onderdelen binnen dezelfde module, tenzij expliciet geëxporteerd.
3. **Herbruikbaarheid**: Modules kunnen zo ontworpen worden dat ze herbruikbaar zijn in verschillende delen van een applicatie, of zelfs in verschillende applicaties.
4. **Lazy Loading**: Modules kunnen "lui" geladen worden, wat betekent dat ze pas geladen worden wanneer ze nodig zijn. Dit kan de initiële laadtijd van de applicatie aanzienlijk verkorten.

Rol ten opzichte van Componenten:

Elke Angular-component behoort tot een Angular-module. De module bepaalt de context waarin een component wordt gecompileerd en uitgevoerd. Componenten die tot dezelfde module behoren, kunnen elkaars functionaliteiten en templates gebruiken. Daarnaast kan een module componenten, services en andere onderdelen van andere modules importeren en gebruiken. Dit alles maakt het mogelijk om complexe applicaties op een modulaire en onderhoudbare manier te bouwen.

In een standaard Angular-toepassing zijn de `AppComponent` en de `AppModule` de basisbouwstenen. Ze dienen als het startpunt van de applicatie.

In Angular zijn de `FormsModule` en de `BrowserModule` twee belangrijke ingebouwde modules die verschillende functionaliteiten bieden:

1. **FormsModule**:
    - De `FormsModule` is verantwoordelijk voor het leveren van de nodige infrastructuur voor het maken en werken met formulieren in Angular. Dit omvat zowel template-gestuurde formulieren als Reactieve formulieren.

- Het stelt ontwikkelaars in staat om gemakkelijk data binding en validatie te implementeren in hun formulieren, waardoor interactie met gebruikersinput efficiënter wordt.
- Met de `FormsModule` kan men bijvoorbeeld `ngModel` gebruiken voor two-way binding, waardoor de synchronisatie tussen het HTML-element en de component eigenschap makkelijker wordt.

2. **BrowserModule**:
    - De `BrowserModule` is een module die specifieke functies voor webbrowsers biedt. Elke Angular-applicatie die in een browser draait, moet deze module importeren.
    - Het bevat belangrijke services die nodig zijn voor het starten en uitvoeren van een Angular-applicatie in een browseromgeving, zoals DOM-sanitization en locatie.
    - Deze module bevat ook de functionaliteit voor het detecteren en initiëren van veranderingen in de componenten van de applicatie, wat essentieel is voor de dynamische natuur van Angular-applicaties.

Kortom, de `FormsModule` is essentieel voor het bouwen en beheren van formulieren in Angular, terwijl de `BrowserModule` noodzakelijke functionaliteiten biedt voor het draaien van een Angular-applicatie in een browser.

Routing

Routing in Angular is het mechanisme dat navigatie tussen verschillende views of componenten in een applicatie mogelijk maakt. Het biedt een manier om de applicatie te structureren als een reeks van verschillende pagina's, waar gebruikers tussen kunnen navigeren. Dit is cruciaal voor het creëren van een Single Page Application (SPA) waarbij de pagina niet volledig herladen hoeft te worden bij navigatie. Via de url kan dan navigatie plaatsvinden. Bvb http://localhost/#/game. In tegenstelling tot klassieke website zal er geen request naar server gaan, maar zal alles lokaal door het framework worden afgehandeld.

De kernaspecten van routing in Angular zijn:

1. **RouterModule**:
    - Dit is de Angular-module die routing functionaliteiten bevat. Het wordt geïmporteerd in de applicatie module met een configuratie die de routes definieert.

2. **Routes Configuratie**:
    - Routes zijn gedefinieerd als een array van objecten waar elk object een route representeert. Elke route koppelt een URL-pad aan een component. Bijvoorbeeld, `{ path: 'home', component: HomeComponent }` zou de `HomeComponent` laden wanneer gebruikers naar `/home` navigeren.
3. **RouterOutlet**:
    - Dit is een directive in de template die als placeholder dient voor waar de content van de huidige route getoond moet worden.
4. **RouterLink**:
    - Een directive die gebruikt wordt in templates om navigatie links te creëren. Bijvoorbeeld, `<a routerLink="/home">Home</a>` creëert een link naar de `home` route.
5. **Router Service**:
    - Een service die programmatische navigatie mogelijk maakt, bijvoorbeeld door de `navigate` en `navigateByUrl` methoden te gebruiken.

Basisvoorbeeld routing

1. *Creëer de Componenten **:
   Laten we beginnen met twee eenvoudige componenten:
   `HomeComponent` en `AboutComponent`.

   home.component.ts

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

   @Component({
     selector: 'app-home',
     templateUrl: './home.component.html'
   })
   export class HomeComponent {}
   ```

   home.component.html

   ```
   <h2>Home Page</h2>
   <p>Welcome to the home page!</p>
   ```

about.component.ts

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

@Component({
  selector: 'app-about',
  templateUrl: './about.component.html'
})
export class AboutComponent {}
```

about.component.html

```
<h2>About Page</h2>
<p>This is the about page.</p>
```

2. **Configureer de App Module**:
   Voeg de componenten toe aan de `AppModule` en importeer de `RouterModule`.

   app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { AppComponent } from './app.component';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    AboutComponent
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot([])
  ],
  providers: [],
```

```
  bootstrap: [AppComponent]
})
export class AppModule { }
```

3. **Implementeer Routing in het AppComponent**:

   app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  routes: Routes = [
    { path: 'home', component: HomeComponent },
    { path: 'about', component: AboutComponent },
    { path: '', redirectTo: '/home', pathMatch: 'full' }
  ];

  constructor() {
    RouterModule.forRoot(this.routes);
  }
}
```

   app.component.html

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

! Om de huidige route te laten weergeven gebruiken we de RouterLinkActive directive.

```
    <a [routerLink]="['twoway']" [routerLinkActive]="['is-dan
```

Services

Angular Services zijn een fundamenteel onderdeel van de Angular-architectuur. Ze zijn ontworpen om **herbruikbare bedrijfslogica of functionaliteit** te bieden die door **verschillende componenten** binnen een Angular-applicatie kan worden gebruikt. Dit bevordert de modulariteit en herbruikbaarheid van code.

Een Angular Service is in essentie een **klasse** met een specifiek doel en verantwoordelijkheid. Het wordt meestal gebruikt voor taken zoals het uitvoeren van **HTTP-verzoeken** naar een server, **het delen van data en functionaliteiten** tussen verschillende componenten, en het **beheren van de applicatiestaat**.

Services in Angular zijn **singleton-objecten**, wat betekent dat er slechts één instantie van een service is tijdens de levensduur van een applicatie. Dit maakt ze uiterst efficiënt voor het beheren van gedeelde bronnen of data.

Door het gebruik van services kunnen **componenten** in Angular zich concentreren op hun **primaire taak**, namelijk het **presenteren van data** en het **afhandelen van gebruikersinteracties**, terwijl de services zorgen voor de logistiek en het beheer van de data of business logica. Dit resulteert in een schone scheiding van zorgen binnen de applicatie, waardoor het onderhoud en testen van de applicatie eenvoudiger wordt.

Kenmerken

1. **Klasse:** Een service in Angular is inderdaad een klasse. Deze benadering zorgt voor duidelijke structuren en maakt gebruik van objectgeoriënteerde programmeerprincipes.
2. **Naamgevingsconventie:** De naam van een service eindigt typisch op 'Service' (bijv. `DataService`, `AuthService`). Dit is niet verplicht, maar wordt sterk aangeraden voor duidelijkheid en consistentie binnen de codebase.
3. **Export Keyword:** De klasse is gemarkeerd met het `export` keyword om deze te kunnen importeren in andere delen van de applicatie.
4. **@Injectable() Decorator:** Deze decorator is essentieel. Het markeert de klasse als een service die geïnjecteerd kan worden, en maakt het mogelijk voor Angular's dependency injection systeem om deze service te beheren en te verstrekken aan componenten of andere services.

5. **Singleton Patroon:** Services in Angular zijn standaard singleton. Dit betekent dat wanneer een service eenmaal is aangemaakt, dezelfde instantie wordt gebruikt door alle componenten die deze nodig hebben.
6. **Dependency Injection (DI):** Services maken vaak gebruik van DI om afhankelijkheden zoals andere services of modules in te voegen. Dit verhoogt de modulariteit en testbaarheid van de applicatie.
7. **Service als Data Provider:** Services worden vaak gebruikt om data op te halen, te bewaren en te manipuleren. Dit kan data zijn die van een server komt, of gedeelde data tussen componenten.
8. **Herbruikbaarheid en Testbaarheid:** Door logica in services te centraliseren, wordt de code herbruikbaar en makkelijker te testen. Dit leidt tot een schonere en meer onderhoudbare codebase.

Een service aanmaken via de CLI

Om een Angular service aan te maken via de Angular CLI, volg je deze stappen:

1. **Open je Command Line Interface:** Zorg dat je in de hoofdmap van je Angular-project bent.
2. **Gebruik de Angular CLI Commando:** Voer het volgende commando in:

   ```
   ng generate service [service-naam]
   ```

   Vervang `[service-naam]` met de gewenste naam van je service. Bijvoorbeeld, als je een service wilt maken voor gebruikersauthenticatie, zou je kunnen typen:

   ```
   ng generate service auth
   ```

   Dit commando genereert twee bestanden: `auth.service.ts` en `auth.service.spec.ts`. Het `.ts` bestand bevat de service klasse, terwijl het `.spec.ts` bestand gebruikt wordt voor testdoeleinden.
3. **Aanpassen van de Service:** Open het gegenereerde `.ts` bestand en voeg je gewenste logica en methoden toe aan de serviceklasse.

(je kan eventueel opteren om alle services onder te brengen in de directory services - ng g s services/demo1)

Basisvoorbeeld van een Angular service

Hier is een eenvoudig voorbeeld van hoe een basis Angular service in TypeScript eruit zou kunnen zien. Dit is een fundamentele structuur van een Angular service, waarin basisfunctionaliteiten worden gedefinieerd die door verschillende componenten in de applicatie hergebruikt kunnen worden.

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

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

  private data: any[] = [];

  constructor() { }

  addData(item: any) {
    this.data.push(item);
  }

  getData() {
    return this.data;
  }

  clearData() {
    this.data = [];
  }
}
```

In dit voorbeeld:

- **Importeren van Injectable:** We importeren `Injectable` uit `@angular/core`, wat nodig is om de service als een injecteerbare afhankelijkheid te markeren.
- **Decorator @Injectable():** De `@Injectable` decorator wordt gebruikt om de service te definiëren. De optie `providedIn: 'root'` zorgt ervoor dat Angular een singleton instantie van deze service creëert die beschikbaar is in de hele applicatie.

- **De Service Klasse:** `DataService` is de naam van de service klasse. Deze klasse bevat drie methoden: `addData` om items toe te voegen, `getData` om alle items op te halen, en `clearData` om alle items te verwijderen.
- **Private Data Eigenschap:** We hebben een private eigenschap `data` die een array is, bedoeld om de data te bewaren die door de service wordt beheerd.

Een component koppelen aan een service via dependency injection

Om een component aan een service in Angular te koppelen, volg je een proces dat bekend staat als dependency injection. Hier is een eenvoudig voorbeeld:

Stel, je hebt een service genaamd `DataService` die data manipuleert. Je wilt deze service gebruiken in een component genaamd `MyComponent`.

1. **Importeer de Service in je Component:**
   Eerst importeer je de service in het componentbestand. Bijvoorbeeld:

   ```
   import { DataService } from './data.service';
   ```

2. **Injecteer de Service via de Constructor van de Component:**
   Vervolgens injecteer je de service in de constructor van de componentklasse. Angular zorgt voor het instantiëren en leveren van de service.

   ```
   export class MyComponent {
     constructor(private dataService: DataService) {}
   }
   ```

   Hierbij declareer je `dataService` als een private eigenschap van `MyComponent`. Angular injecteert de singleton-instantie van `DataService` in deze eigenschap.

3. **Gebruik de Service in de Component:**
   Nu kun je de methoden van de service binnen je component gebruiken. Bijvoorbeeld, als
   `DataService` een methode `getData()` heeft, kun je deze aanroepen in je component:

   ```
   ngOnInit() {
     this.dataService.getData().subscribe(data => {
       // Doe iets met de data
   ```

```
    });
  }
```

In dit voorbeeld wordt de `getData()` methode van de `DataService` aangeroepen wanneer de component wordt geïnitialiseerd (`ngOnInit`). De component kan nu gebruik maken van de functionaliteiten die de service biedt, zoals het ophalen, bewaren of manipuleren van data. Dit is een typisch patroon in Angular om componenten en services met elkaar te verbinden.

Bewaren van State

Laten we een voorbeeld nemen waarbij een Angular-service wordt gebruikt om een gedeelde string-waarde ('state') te beheren, die door twee componenten, `ComponentA` en `ComponentB`, wordt gedeeld.

Eerst definiëren we de service, `SharedStateService`:

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

@Injectable({
  providedIn: 'root'
})
export class SharedStateService {
  private sharedState: string = '';

  setSharedState(state: string) {
    this.sharedState = state;
  }

  getSharedState(): string {
    return this.sharedState;
  }
}
```

In `ComponentA`, stel je de staat in:

```
import { Component } from '@angular/core';
import { SharedStateService } from './shared-state.service';

@Component({
```

```
  selector: 'app-component-a',
  template: `<button (click)="updateState()">Set State</butto
})
export class ComponentA {
  constructor(private sharedStateService: SharedStateService)

  updateState() {
    this.sharedStateService.setSharedState('Nieuwe State');
  }
}
```

In `ComponentB`, lees je de staat:

```
import { Component } from '@angular/core';
import { SharedStateService } from './shared-state.service';

@Component({
  selector: 'app-component-b',
  template: `<div>Current State: {{ currentState }}</div>`
})
export class ComponentB {
  currentState: string;

  constructor(private sharedStateService: SharedStateService)
    this.currentState = this.sharedStateService.getSharedStat
  }
}
```

In dit voorbeeld:

- `SharedStateService` beheert een string-waarde (`sharedState`).
- `ComponentA` heeft een methode om de staat te wijzigen via de service.
- `ComponentB` leest de huidige staat uit de service.

Dit zorgt ervoor dat beide componenten dezelfde staat delen via de service. Wanneer `ComponentA` de staat wijzigt, kan `ComponentB` deze wijziging weergeven door de nieuwe waarde uit de service te halen.

Aanroepen van externe (http) services: HttpClient

We gebruiken hiervoor de Angular HttpClientModule en de HttpClient klasse

- Kan gebruikt worden om data op een asynchrone wijze op te halen (GET) of weg te schrijven (POST/PUT)
- Aan de hand van de service URL
- Veelal geven de services json-data terug. Deze zal door de HttpClient worden omgezet naar objecten, zodat we de data in code kunnen uitlezen en verwerken
- De HttpClientModule moet worden toegevoegd aan de AppModule
- De HttpClient zal via Dependency Injection in de service worden geïnjecteerd
- De HttpClient heeft een 'get' method om een GET request te doen met de service URL als parameter

app.module.ts:

```
    import { HttpClientModule } from '@angular/common/http';
    ... imports: [..., HttpClientModule, ...]
```

Laten we een voorbeeld bekijken waarin een Angular service gebruik maakt van `HttpClient` om data van een externe bron op te halen, en een component die zich abonneert op de resulterende JSON-data via een Observable.

1. **Service met HttpClient:**
   Eerst definieer je een service die `HttpClient` gebruikt om data op te halen.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class DataService {
  private apiUrl = '<https://api.example.com/data>'; // Ve

  constructor(private http: HttpClient) {}

  fetchData(): Observable<any> {
```

```
    return this.http.get(this.apiUrl);
  }
}
```

In deze service:

- We importeren en injecteren `HttpClient`.
- De `fetchData` methode gebruikt `HttpClient` om een GET-verzoek naar de API te sturen.
- Het resultaat is een Observable die de resulterende JSON-data zal verstrekken.

2. **Component die zich abonneert op de Service:**
   Vervolgens maak je een component die deze service gebruikt en zich abonneert op de Observable om de data te ontvangen.

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-data-consumer',
  template: `<div *ngIf="data">Data: {{ data | json }}</di
})
export class DataConsumerComponent implements OnInit {
  data: any;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.dataService.fetchData().subscribe(receivedData =>
      this.data = receivedData;
    });
  }
}
```

In deze component:

- In de `ngOnInit` lifecycle hook, roepen we de `fetchData` methode van `DataService` aan.

- We abonneren ons op de Observable die door deze methode wordt geretourneerd.
- Zodra de data binnenkomt, wordt deze toegewezen aan de lokale `data` variabele, die vervolgens kan worden gebruikt in de component's template.

Deze setup laat zien hoe je een externe API kunt aanroepen via een service in Angular en hoe je een component kunt abonneren op de resulterende data met behulp van Observables.

Component lifecycle

- Laatste stap die nodig is om de service te kunnen afwerken is het kennen van de component lifecycle
- Elk Angular component heeft immers een levenscyclus
- constructor, OnInit, OnChanges, OnChanges, OnDestroy
- Angular maakt het mogelijk om op elke fase in te haken
- Dit kan door het implementeren van de juiste interface in de component

```
    export class LeegstandLijstComponent implements OnInit, O
        constructor() {}
        ngOnInit() {}
        ngOnChanges(changes: SimpleChanges): void {}
        ngOnDestroy(): void {}
```

Aanroepen van een service: OnInit. Waarom? In de constructor geen code aanroepen die kan mislukken. De OnInit fase wordt aangeroepen onmiddellijk nadat een component wordt aangemaakt.

```
    export class LeegstandLijstComponent implements OnInit {

        lijst: ILeegstand[];
        constructor(private service: LeegstandService) {}
        ngOnInit() {
            this.service.subscribe(d => this.lijst = d);
        }
```

Ter info:

De `ngOnChanges` lifecycle hook in Angular biedt waardevolle functionaliteit voor het detecteren van en reageren op veranderingen in input properties van een component. Deze hook wordt aangeroepen wanneer de input properties van een component veranderen. Het is bijzonder nuttig in situaties waarin een component dynamisch moet reageren op veranderingen in de data die het ontvangt van zijn oudercomponent.

Hier is een eenvoudige illustratie van hoe `ngOnChanges` kan worden gebruikt:

Stel, je hebt een component `ChildComponent` die een input property `@Input() data: string;` heeft. Je wilt dat `ChildComponent` bepaalde acties uitvoert wanneer de waarde van `data` verandert.

```
import { Component, Input, OnChanges, SimpleChanges } from '@

@Component({
  selector: 'app-child',
  template: `<h3>{{ data }}</h3>`
})
export class ChildComponent implements OnChanges {
  @Input() data: string;

  ngOnChanges(changes: SimpleChanges) {
    if (changes['data']) {
      console.log('Data gewijzigd:', changes['data'].currentV
      // Voer hier aanvullende acties uit wanneer data verand
    }
  }
}
```

In dit voorbeeld:

- De `ChildComponent` heeft een input property `data`.
- Wanneer de waarde van `data` verandert (bijvoorbeeld door een oudercomponent), wordt `ngOnChanges` geactiveerd.
- Binnen `ngOnChanges`, kun je controleren welke input properties zijn veranderd en daarop reageren. In dit geval loggen we gewoon de nieuwe waarde van `data`.

Dit is vooral handig in gevallen waarin de reactie op de verandering meer inhoudt dan alleen het opnieuw renderen van de view, bijvoorbeeld wanneer je een verzoek

naar een server moet sturen of complexe logica moet uitvoeren als reactie op de verandering.

Goed alternatief voor observables: promises

- Promises zijn een goed alternatief voor observables
- Een observable kan immers worden omgezet via de .toPromise() method
- Voordeel van promises is dat we gebruik kunnen maken van async/await
- Hiermee krijgen we beter gestructureerde broncode (zeker indien meerdere calls na elkaar)

Het gebruik van promises in plaats van observables heeft enkele voordelen, afhankelijk van de specifieke vereisten van je toepassing. Hier zijn enkele belangrijke voordelen:

1. **Eenvoud en Begrijpelijkheid:** Promises zijn vaak eenvoudiger en directer voor ontwikkelaars die niet vertrouwd zijn met de reactieve programmeerpatronen van RxJS en observables.
2. **Eénmalige Waarde:** Promises zijn ontworpen om een enkele waarde te behandelen. Ze zijn ideaal voor eenvoudige asynchrone operaties waar je slechts één keer data ophaalt of een enkele actie uitvoert.
3. **Automatische Foutafhandeling:** Foutafhandeling met promises kan worden gestroomlijnd met `.catch()` blokken, wat kan leiden tot helderdere foutafhandelingslogica.
4. **Native Ondersteuning in JavaScript:** Promises zijn een onderdeel van de JavaScript-taal zelf, wat betekent dat ze geen extra bibliotheken vereisen.

Nu, om het voorbeeld van de service en component met een promise in plaats van een observable aan te passen:

1. **Service met HttpClient en Promise:**
   De service gebruikt `HttpClient` maar converteert de Observable naar een Promise met de `.toPromise()` methode.

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

@Injectable({
```

```
  providedIn: 'root'
})
export class DataService {
  private apiUrl = '<https://api.example.com/data>'; // Ve

  constructor(private http: HttpClient) {}

  fetchData(): Promise<any> {
    return this.http.get(this.apiUrl).toPromise();
  }
}
```

2. **Component die een Promise gebruikt:**
   De component roept de service aan en handelt de Promise af.

```
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-data-consumer',
  template: `<div *ngIf="data">Data: {{ data | json }}</di
})
export class DataConsumerComponent implements OnInit {
  data: any;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.dataService.fetchData().then(receivedData => {
      this.data = receivedData;
    }).catch(error => {
      console.error('Er is een fout opgetreden', error);
    });
  }
}
```

In deze aangepaste component:

- We gebruiken `.then()` om de ontvangen data te behandelen.

- `.catch()` wordt gebruikt voor foutafhandeling.

Dit aangepaste voorbeeld laat zien hoe je met Promises kunt werken voor eenvoudige, eenmalige asynchrone operaties.