Angular
Data Binding

1 Einfachste App

Es soll noch einmal mit der einfachsten App begonnen werden.
Am besten die App im Debug-Modus wie im vorherigen Dokument beschrieben starten.

main.ts muss praktisch nie geändert werden.

1.1 app.component.html

Allen Code löschen und durch folgende Zeile ersetzen:

```
<h1>Hello World</h1>
```

1.2 app.module.ts

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

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


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

Für die Verwendung von [(ngModel)] (siehe unten) muss das **`FormsModule`** importiert werden.

1.3 app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
  title = 'angular13';

  ngOnInit(): void {
    this.title = 'Hello Angular';
  }
}
```

2 Data Binding

Die Funktionsweise von Angular soll an einem ganz einfachen Beispiel demonstriert werden. Dabei wird noch (fast) kein eigener Typescript-Code programmiert – es spielt sich alles im HTML-File ab.

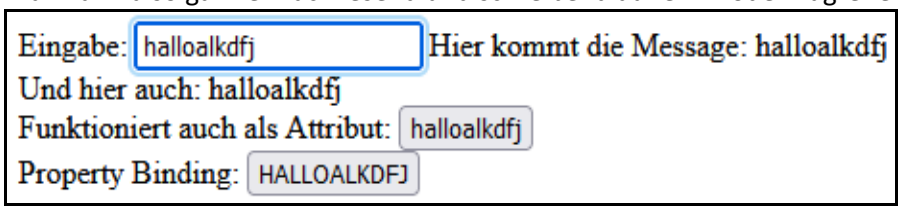
```
app > <> app.component.html > ...
  Go to component
  Eingabe: <input type="text"
    2 [(ngModel)]="message" />

  Hier kommt die Message: {{message}}
  <div>Und hier auch: {{message}}</div>  3
  <div>Funktioniert auch als Attribut: <input type="button"
        value="{{message}}" /></div>

  <div>Property Binding: <input type="button"
    4 [value]="message.toUpperCase()" /></div>
```

```
export class AppComponent {
  title = 'hello-angular';
  message = 'hallo';  1
}
```

Dabei erkennt man schon die wesentlichen Elemente einer Angular-App:

1	**Model**	Die Variable message muss als Instanzvariable der Component-Klasse definiert werden.
2	**[(ngModel)]**	Mit [(ngModel)] kann man auf eine Variable des Modells **lesend** und **schreibend** zugreifen.
3	**{{…}}**	Damit verweist man auf eine Property des Modells, das von der zuständigen Component verwaltet wird. Diese Variante nennt man **Interpolation**.
4	**[prop]="…"**	Sehr ähnlich funktioniert **Property binding**. Dabei wird die DOM-Property in eckige Klammern gesetzt. Der Wert ist ein **Typescript** -Ausdruck.

Man kann also ganz einfach lesend und schreibend auf ein Modell zugreifen.



2.1 Vergleich MVVM/WPF

Die Bindung erfolgt bekanntlich bei MVVM auf folgende Weise:

Model	**private string _message = "hallo";** **public string Message** **{** ** get => message;** ** set** ** {** ** message = value;** ** NotifyPropertyChanged(nameof(Message));** ** }** **}**
[(ngModel)]	**<TextBox Text="{Binding Message,** **Mode=TwoWay,** **UpdateSourceTrigger=PropertyChanged}" />**
{{…}}	**<Label Content="{Binding Message}" />**

Zuordnung View←→Model

Angular	In der Klasse:
	`@Component({`
	`  selector: 'app-root',`
	`  templateUrl: './app.component.html',`
	`  styleUrls: ['./app.component.scss']`
	`})`
	`export class AppComponent implements OnInit {`

MVVM	Im XAML (für XAML-Editor):
	`<Window x:Class="MvvmDemo.MainWindow"`
	`…`
	`xmlns:vm="clr-namespace:ArtistViewModelLib;`
	`assembly=ArtistViewModelLib"`
	`d:DataContext="{d:DesignInstance vm:ArtistViewModel }"`
	`…>`
	In der Klasse (zur Laufzeit) – Beispiel ohne Dependency Injection:
	`private void Window_Loaded(object sender, RoutedEventArgs e)`
	`{`
	`  var db = new ArtistContext();`
	`  var viewModel = new ArtistViewModel(db);`
	`  DataContext = viewModel;`
	`}`

2.2 One-Way Interpolation

One-Way Databinding wird in Angular durch Angabe von ***{{MyVariableName}}*** ermöglicht. Derartige Codesequenzen im HTML entsprechen also dem **View**-Teil des MVC-Patterns.
Dabei ist es egal, ob man den Wert der Variable direkt ausgibt (als Inhalt eines) oder als Attributswert verwendet (siehe obiges <input>-Element).
Der **Kontext** ist immer jene Komponente (bzw. auch Direktive), in der der Ausdruck verwendet wird. Im Beispiel wird also **`message`** als Property in der Klasse **`AppComponent`** gesucht.

2.3 One-Way Property Binding

Setzt man eine Element-Property in **eckige Klammern**, wertet Angular den angegebenen **Typescript-Ausdruck** aus und weist ihn auf die Property zu. Für den Kontext gilt dabei dasselbe wie für Interpolation.
Im Prinzip sind Interpolation und Property Binding gleich. Welche Variante man wählt ist mehr oder weniger Geschmackssache.

2.4 Two-Way Databinding

Databinding in beide Richtungen – also sowohl lesend als auch schreibend – wird in Angular durch Angabe des Attributs **`[(ngModel)]="myVariableName"`** ermöglicht. Das entspricht somit dem **Controller**-Teil des MVC-Patterns. Die Struktur **`[()]`** bezeichnet man auch als „banana in the box“.
Hinweis: laut HTML-Standard sind Zeichen wie **`[`** und **`(`** in Attributsnamen erlaubt. Das wird in Angular ausgenutzt, um schreibenden Zugriff zu kennzeichnen.

2.5 templateUrl

Für kurze HTML-Sequenzen ist die Notation mit einem Template String unter Umständen sinnvoll und übersichtlich, weil alles in einer Datei steht. Für längeren HTML-Code ist das aber eher nicht mehr geeignet, weil man auch Intellisense verliert. Die Property heißt dann **`template`** statt **`templateUrl`**:

```
@Component({
  selector: 'app-root',
  template: `
    Eingabe: <input type="text" [(ngModel)]="message" />
    Hier kommt die Message: {{message}}
  `,
  styleUrls: ['./app.component.scss']
})
```

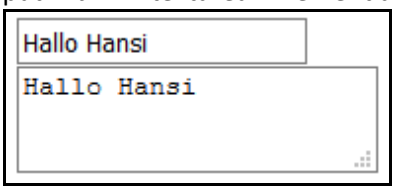
3 Eingabe-Elemente

Mit diesem Wissen kann man schon die grundlegenden HTML-Elemente benutzen. Dabei muss man noch keinen Typescript-Code schreiben!

3.1 Text/Textarea

Für Texteingaben gibt man **[(ngModel)]** für das <input>- bzw. <textarea>-Element an.

```
<input type="text" [(ngModel)]="myText" />
<br />
<textarea [(ngModel)]="myText"></textarea>
```



Die Variable sollte man in der Klasse definieren, muss man aber nicht unbedingt (Wert ist dann zu Beginn undefined und wird somit als Leerstring dargestellt).

```
export class AppComponent {
  message = 'aaaa';
  myText = 'Hallo Hansi';
}
```

3.2 Combobox

Für Comboboxen gibt man das **[(ngModel)]**-Attribut im **<select>**-Element an. Die Werte, die auf die Modellvariable geschrieben werden, müssen im **value**-Attribut angegeben werden.

```
<select [(ngModel)]="myCombo">
  <option value="aaaa">aaaa</option>
  <option value="bbbb">bbbb</option>
  <option value="cccc">cccc</option>
</select>
Selektiert: {{myCombo}}
```

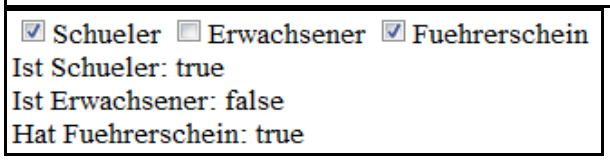
```
myCombo = 'cccc';
```

3.3 Checkbox

Für Checkbox-Elemente verhält es sich praktisch genau sie, wie für Textfelder. Der Modellwert ist jedoch vom Typ boolean.

```
<input type="checkbox" [(ngModel)]="isStudent" />Schueler
<input type="checkbox" [(ngModel)]="isAdult" />Erwachsener
<input type="checkbox" [(ngModel)]="hasLicence" />Fuehrerschein
<br />
Ist Schueler: {{isStudent}} <br />
Ist Erwachsener: {{isAdult}} <br />
Hat Fuehrerschein: {{hasLicence}} <br />
```

```
isStudent = true;
isAdult = false;
hasLicence = true;
```



3.4 Radiobuttons

Auch für Radioboxes muss man nicht kompliziert denken.

```
<input type="radio" name="country" [(ngModel)]="myCountry" value="A" />Austria
<input type="radio" name="country" [(ngModel)]="myCountry" value="D" />Germany
<input type="radio" name="country" [(ngModel)]="myCountry" value="I" />Italy
```

```
myCountry = 'A';
```

Austria Germany Italy
Land: A

3.5 *ngFor

Ein weiteres Highlight von Angular ist die Möglichkeit, clientseitig aus Templates HTML-Code erzeugen zu können. Der DOM-Subtree, der wiederholt werden soll, wird mit dem Attribut **`*ngFor`** markiert. *ngFor ist (wie auch ngModel) eine sogenannte Direktive.
Wie der Name vermuten lässt, muss man *ngFor ein **Array** von Werten/Objekten mitgeben, das durchlaufen werden soll. Damit wird angegeben, dass der HTML-Block als Template zu sehen ist, der pro Eintrag in der Collection in den DOM eingehängt wird. Innerhalb des Templates greift man dann wie beim **One-Way-Databinding** auf die jeweiligen Werte zu, also mit **`{{…}}`**.
Format: **`*ngFor="let item of collection"`**
Die zu durchlaufende Collection muss vom Javascript-Typ Iterable sein (daher auch das **`of`** und nicht **`in`**).
Die folgenden Beispiele sollten das verständlich machen.

3.5.1 Liste

Bei einer Liste wird der -Tag wiederholt, daher muss hier *ngFor stehen.

```
<ul>
  <li *ngFor="let person of ['Hansi','Susi','Fritzi','Gerti']">
    {{person.toUpperCase()}}
  </li>
</ul>
```

- HANSI
- SUSI
- FRITZI
- GERTI

3.5.2 Tabelle

Bei einer Tabelle sollen Tabellenzeilen erzeugt werden, daher wird *ngFor bei <tr> notiert.

```
<table>
  <tr>
    <th>#</th>
    <th>Name</th>
  </tr>
  <tr *ngFor="let person of ['Hansi','Susi','Fritzi','Gerti']; let index=index">
    <td>{{index+1}}</td>
    <td>{{person.toUpperCase()}}</td>
  </tr>
</table>
```

#	Name
1	HANSI
2	SUSI
3	FRITZI
4	GERTI

Angular kennt einige interne Variable, die automatisch mit Werten belegt werden. Eine dieser Variablen ist **`index`**, die als Zählvariable beim *ngFor-Durchlauf fungiert.

3.5.3 Lokale Variablen

Neben **`index`** kann man noch weitere lokale Variablen innerhalb von *ngFor verwenden, die alle vom Typ boolean sind:

- **`first`**: erstes Element?
- **`last`**: letztes Element?
- **`even`**: gerader Index?
- **`odd`**: ungerader Index?

3.5.4 Tabelle mit Objekten

Auf die gleiche Weise kann man Arrays mit Objekten durchlaufen und damit Code erzeugen lassen.

```
<table>
  <tr>
    <th>#</th>
    <th>Vorname</th>
    <th>Nachname</th>
    <th>Name</th>
  </tr>
  <tr *ngFor="let person of [{firstname:'Hansi',lastname:'Huber'},
                             {firstname:'Susi',lastname:'Berger'},
                             {firstname:'Fritzi',lastname:'Gruber'},
                             {firstname:'Gerti',lastname:'Mueller'}];
              let index=index">
    <td>{{index+1}}</td>
    <td>{{person.firstname}}</td>
    <td>{{person.lastname}}</td>
    <td>{{person.lastname}} {{person.firstname}}</td>
  </tr>
</table>
```

#	Vorname	Nachname	Name
1	Hansi	Huber	Huber Hansi
2	Susi	Berger	Berger Susi
3	Fritzi	Gruber	Gruber Fritzi
4	Gerti	Mueller	Mueller Gerti

3.5.5 Combobox mit *ngFor

Die Werte der Combobox kann man auch über *ngFor aus einem Array zuweisen:

```
<select [(ngModel)]="country">
  <option *ngFor="let x of [{ id: 'A', descr: 'Austria' }, { id: 'D', descr: 'Germany' }]"
          [value]="x.id">
    {{x.descr}}
  </option>
</select>
{{country}}
```

Spätestens jetzt sollte man sehen, dass die Angabe der Daten (hier die Personenliste) im Code keine gute Idee sind. Wir werden also unsere eigenen Modelle programmieren wollen/müssen. Das führt uns zu den Komponenten-Klassen.

3.6 ngValue

Der Wert für [value] kann immer nur ein String sein.
Braucht man hingegen ein Objekt als „key“, muss man **`[ngValue]`** benutzen

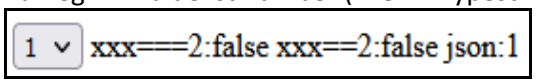
```
<select [(ngModel)]="country">
  <option *ngFor="let x of [{ id: 'A', descr: 'Austria' }, { id: 'D', descr: 'Germany' }]"
          [ngValue]="x">
    {{x.descr}}
  </option>
</select> {{country|json}}
```

Das ist auch in Situationen wichtig, wo man es vielleicht nicht gleich erwarten würde:

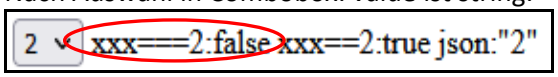
```
<select [(ngModel)]="xxx">
  <option *ngFor="let x of [1,2,3,4]"
          [value]="x">
    {{x}}
  </option>
</select>
xxx===2:{{xxx===2}} xxx==2:{{xxx==2}} json:{{xxx|json}}
```

```
xxx = 1;
```

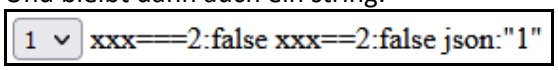
Zu Beginn: value ist number (wie im Typescript definiert):



Nach Auswahl in Combobox: value ist string:



Und bleibt dann auch ein string:



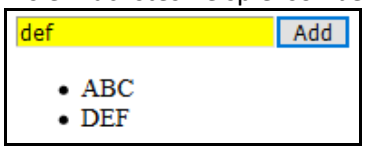
Hinweis: bei <input>-Elementen funktioniert [ngValue] nicht, würde aber auch keinen Sinn machen – ein <input> ist immer eine Texteingabe.

4 Events

So wie man mit **[(ngModel)]** eine Variable des Modells verändern kann, kann man mit Events Funktionen des Modells aufrufen. Dabei ist man bei Angular den Weg gegangen, dass man keine neuen Events erfindet (wie z.B. **ng-click** in Angular 1) sondern die **bestehenden Events** einfach mit runden Klammern **()** umgibt und dahinter eine Komponenten-Funktion registriert, die daher dann auf die Klassenvariablen zugreifen kann.
Da die existierenden Events benutzt werden, aber eben mit **()**, muss man nichts Neues lernen und kann dann alle Events benutzen, wenn man das Grundprinzip einmal verstanden hat.

4.1 Beispiel click

Als einfachstes Beispiel soll bei Klick auf einen Button der Inhalt eines Textfeldes in eine Liste eingefügt werden:



Das Event heißt **click**, daher muss der Eventhandler in Angular mit **(click)** notiert werden:

```
<input type="text"
       [(ngModel)]="itemToAdd" />
<button (click)="addItem()">Add</button>
<ul>
  <li *ngFor="let item of items">
    {{item.toUpperCase()}}
  </li>
</ul>
```

Die Funktion, die bei (click) angegeben wird, muss in der Komponente implementiert werden:

```
export class AppComponent {
  items: string[] = [];
  itemToAdd = 'abc';

  addItem(): void {
    this.items.push(this.itemToAdd);
  }
}
```

Eine Modell-Funktion ist also eine ganz normale Property der Komponente, deren Wert eine Funktion ist.
Hinweise:

- Innerhalb der Funktion kann man auf alle Properties der Komponente zugreifen.
- Dabei darf man **this** nicht weglassen (wie z.B. in C#)
- Man könnte im HTML beim Handler von **(click)** auch direkt auf eine Klassenvariable zugreifen und diese ändern. Das wird aber nicht empfohlen, weil somit die Trennung von View und Controller verwaschen wird.
- Die Funktion kann auch Parameter haben (siehe später)
- Es ginge auch ohne **ngModel** (auch dazu später mehr)

4.2 Vergleich MVVM/WPF

Auch hier ist die Angular-Schreibweise etwas kürzer – obiger Code würde in MVVM so aussehen:
Klasse:

```
public ICommand AddItem => new RelayCommand<string>( x => Items.Add(ItemToAdd) );
```

View:

```
<Button Content="Add" Command="{Binding AddItem}"/>
```

Hauptunterschied:
In Angular kann man alle verfügbaren Javascript-Events binden, in MVVM nur Click-Events (außer man verwendet Behaviors).