Angular: Einführung

- Verstehen wozu Angular eingesetzt wird und welche Eigenschaften es hat
- Welche Voraussetzungen müssen gegeben sein, um das erste Web-Projekt zu erstellen, Projektstruktur verstehen
- Wichtige Dateien für den Startvorgang und ihre Bedeutung verstehen
- Grundlagen von TypeScript beherrschen
    - Typen und Variablendeklarationen
    - Klassen mit Getter- und Settermethoden und Konstruktoren
    - Verschiedene Arten von Interfaces
    - Mit Arrow-Funktionen bzw. Lambda-Ausdrücken umgehen
- Eigene Komponenten erstellen und mit ihren Bestandteilen arbeiten können
- Die Vorteile von Interpolation, Property- und Event-Binding, Elementreferenzen, Strukturdirektiven und Safe-Navigation-Operator erkennen und mit ihnen umgehen können
- Einfache Formulare erstellen können
- Eigene Komponenten in Hierarchien verwenden und Werte an Komponenten übergeben bzw. von Komponenten erhalten können
- Responsives Web-Design mit Angular Material und Flex-Layout realisieren können

Literaturhinweis

Angular, das umfassende Handbuch, Christoph Höller, Rheinwerk Verlag, ISBN 978-3-8362-8243-7

Was ist Angular?

- Framework zur Entwicklung von Single-Page-Applikationen
- Web-Applikation wird vom Browser nachgeladen und dort ausgeführt
- Entwicklungswerkzeug sowohl für Webapplikationen als auch für native Apps für Mobil- und Desktopgeräte (Android, iOS, Windows)
- Modular aufgebaut, wiederkehrende Standardaufgaben werden durch Werkzeuge durchgeführt
- Komponentenbasierte Entwicklung
- Open Source
- Programmiersprache TypeScript

Voraussetzungen

1. **Visual Studio Code** als Editor
2. **Node.js** Laufzeitumgebung zur Ausführung von JavaScript auf dem Server
3. **Node Package Manager** (**NPM**) läuft auf Node.js und installiert dort Anwendungen, verwaltet Abhängigkeiten in einer Node-Anwendung
   Beispielaufruf `npm --version`
4. **Angular Command Line Interface** (**CLI**) beinhaltet Vorlagen und Befehle für alle wiederkehrenden Aufgaben in Angular-Anwendung (Anlegen, Deployment, ...)
   Beispielaufruf `ng version`
5. **Google Chrome** zum Testen

Das erste Projekt

Projekt erstellen mit Angular CLI

```
$ ng new K01TestA --prefix=ta --routing=false
  --style=scss
$ cd K01TestA
$ ng version
```

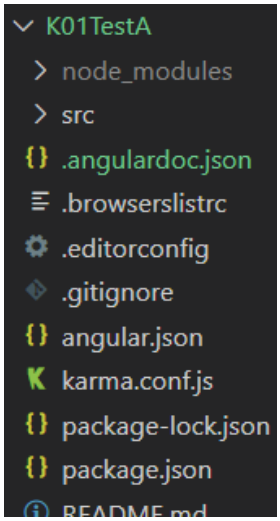
- Projekt erhält *Präfix* `ta` (siehe hinten)
- Kein *Routing* wird für Projekt vorgesehen (siehe Kapitel über Routing)
- Dateinamenserweiterung oder verwendeter Preprozessor für Style-Dateien (siehe hinten *„Responsives Web-Desing mit Angular Material“*)

HINWEIS: Zum Kopieren von Angular-Anwendungen

Ordner `node_modules` muss nicht kopiert werden. Wurden die restlichen Dateien an einem anderen Ort eingefügt, können die notwendigen NPM-Pakete folgendermaßen nachinstalliert werden:

```
$ npm install
```

Die Projektstruktur

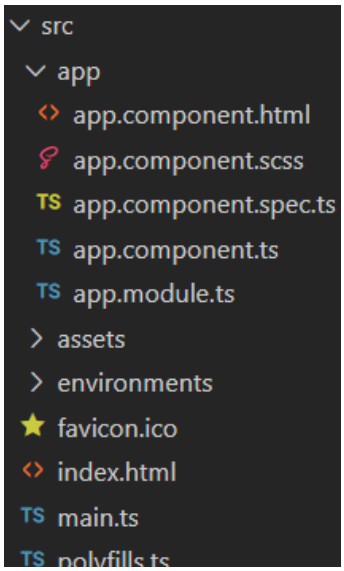


`node_modules`
Installierte NPM-Pakete

`src`
Code der Anwendung

`angular.json`
zentrale Konfigurationsdatei der Anwendung

`package.json`
Konfigurationsdatei des NPM und Abhängigkeiten



`app`
beinhaltet alle Module, Komponenten, Klassen, usw.

`assets`
Ordner für Dateien (Bilder, Videos, Sounds, usw.)

Wichtige Dateien für den Startvorgang

- `index.html`
- `main.ts`
- `app.module.ts`
- `app.component.ts`

HINWEIS: `.spec.ts` enthalten Unit-Tests

Datei index.html

```
<!doctype html>
  …
<body>
  <ta-root>Die Anwendung wird geladen...</ta-root>
</body>
```

Einstiegsseite beim Aufruf im Browser. Enthält <ta-root>-Tag durch welches Anwendung eingebunden wird (siehe hinten)

Datei main.ts

```
…
import { AppModule } from './app/app.module';
…
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(err));
```

Startet die Anwendung mit der zentralen Komponente AppModule

Datei app.module.ts Hauptmodul der Anwendung

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

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

Modul

einfache Klasse die mit einem *Decorator* ergänzt wurde. Dieser hängt Metadaten an die Klasse

declarations muss alle selbsterstellten Komponenten der Anwendung enthalten (siehe hinten)

Fasst alle Teile der Anwendung zusammen und legt Komponente AppComponent fest die sichtbar gemacht wird

Dateien app.component.ts und app.component.html

```
…
@Component({
  selector: 'ta-root',
  templateUrl: './app.component.html',
  styles: ['./app.component.scss']
})
export class AppComponent {
  public title: string;

  constructor() {
    this.title = 'Angular';
  }
}
```

```
<h1>
  Hallo {{ title }}!
</h1>
```

{{ Interpolation }}

- *Root-Komponente* der Anwendung, an einen *CSS-Selector* gebunden, Layout wird in Template-Datei ausgelagert
- Wäre `title private` dann könnte in der Template-Datei nicht auf die *Member-Variable* zugegriffen werden. `public` könnte auch weggelassen werden, weil `public` *Standardeinstellung* ist
- Damit einer Variable auch `null` oder `undefined` zugewiesen werden kann, müsste in `tsconfig.json` die *TypeScript-Compiler Option* `strict` auf `false` gesetzt werden.

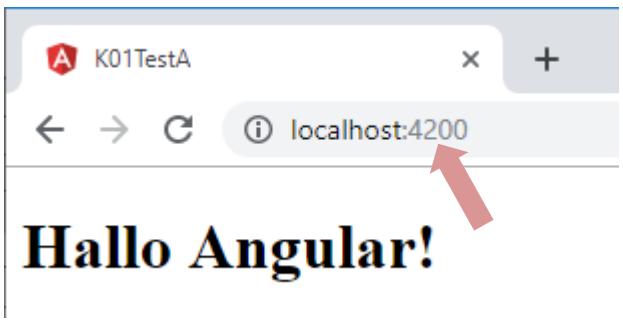
  Momentane Checks können mit `title!` unterbunden werden.

  Dadurch werden Sie gezwungen, eine Variable entweder bei ihrer Deklaration oder spätestens im Konstruktor zu initialisieren und im Code Checks auf `null` oder `undefinied` durchzuführen

Projekt Deployment

```
$ ng serve oder $ nmp start
```

Änderungen am Quellcode haben sofortige Aktualisierung im Browser zur Folge



TypeScript

TypeScript von Microsoft entwickelt ermöglicht starke Typisierung und ist Obermenge von JavaScript

```
var count: number = 3;

let name: string = 'Sepp';

const open: boolean = true;
open = false;
```

Variable innerhalb *Funktion* gültig

Variable innerhalb *Block* gültig

Konstante innerhalb *Block* gültig

number

```
let age: number = 5;
let height: number = 10.5;
```

boolean

```
let male: boolean = true;
```

string

```
let name: string = 'John';
let output: string =
  `${male?'Mr.':'Mrs.'} ${name} is ${age} years old`;
```

ACHTUNG Template-String: schräge Hochkommas `

array

Untypisiert

```
let differentValues = [
  0,                                    // Zahl
  { firstname: 'John' },                // Objekt
  function() { console.log('John'); }   // Funktion
];
```

Typisiert

```
let fibonacci: Array<number> = [0, 1, 1, 2, 3, 5, 8];
let fibonacci2: number[] = [0, 1, 1, 2, 3, 5, 8];
```

any

```
let response: any;
```

Kann jeden beliebigen Datentyp annehmen, Compiler unterlässt jegliche Typprüfung

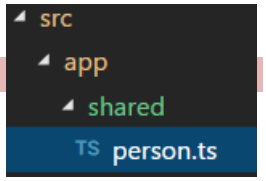
object

```
let myObject: object = { name: 'John', age: 18 };
myObject = 42;
```

Nimmt Objekte beliebigen Typs auf, aber nicht primitive Daten

Klassen

```
$ ng generate class shared/person
```



Explizite Getter-/Settermethoden für Eigenschaften

```
export class Person {
  private _name: string;

  constructor(name: string) {
    this._name = name;
  }

  set name(name: string) {
    this._name = `${this._name} ${name}`;
  }

  get name(): string {
    return this._name;
  }
}
```

public, private, protected, static, readonly, public ist Standardeinstellung

Statische Elemente sind nur über Klasse aufrufbar und sind im Template nicht verfügbar

Membervariable _name muss unterschiedlichen Namen als *Eigenschaft* name haben ansonsten kann nicht zwischen beiden unterschieden werden

Verwendung

```
import { Person } from './shared/person';
…
const john: Person = new Person('John');
john.name = 'Johnny';
console.log(`Name: ${john.name}`); // liefert Name: John Johnny
```

Eine ts-Datei kann auch *mehrere Klassen* aufnehmen von denen nur bestimmte zur Verfügung gestellt werden (export und import)

```
export class Athlete extends Person {
  city: string;
  private fitnesslevel: number = 1;

  constructor(name: string, city: string, fitnesslevel?: number) {
    super(name);
    this.city = city;
    if (fitnesslevel !== undefined)
      this.fitnesslevel = fitnesslevel;
  }

  train(workoutHardness: number): number {
    return this.fitnesslevel += workoutHardness;
  }
}
```

Nur ein Konstruktor pro Klasse möglich

? optionaler Parameter

Wegen *TypeScript-Compiler Option* `strict = true` muss dafür gesorgt werden, dass `this.fitnesslevel` nie `null` werden kann

Verwendung

```
const john: Athlete = new Athlete('John', 'New York');
console.log(john.train(2));          // liefert 3
```

Kurzschreibweise

```
export class Athlete extends Person {

  constructor(
    name: string,
    public city: string,
    private fitnesslevel: number = 1
  ) {
    super(name);
  }
  …
}
```

Setzt Standardwert wenn Parameter nicht übergeben wird

ohne	Konstruktor erhält lediglich Parameter der in ihm verarbeitet werden muss
`public`	öffentliche Membervariable wird in Klasse definiert
`private`	private Membervariable wird in Klasse definiert

```
let alf: Athlete = new Athlete('Alf', 'New York');
console.log(alf.name);               // liefert Alf
console.log(alf.city);               // liefert New York
console.log(alf.fitnesslevel);       // liefert Compilerfehler
```

Interfaces

```
interface ContactInterface {
  firstname: string;  lastname: string;
  age?: number | string;                                    Union-Typ
}

const contactInterface: ContactInterface =
  { firstname: 'Max', lastname: 'Smith', age: '18' };          // *)

class Contact implements ContactInterface {
  public firstname: string;
  public lastname: string;
  public age?: number | string;

  constructor(contact: ContactInterface) {
    this.firstname = contact.firstname;
    this.lastname = contact.lastname;
    if (contact.age !== undefined)
      this.age = contact.age;
  }
}

const contact = new Contact(contactInterface);
console.log(contact);                                      // liefert *)
```

Membervariablen müssen in der implementierenden Klasse redefiniert werden

Duck-Typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.[1]

```
const contact1 = new Contact(
  { firstname: 'Lara', lastname: 'Croft' });
const contact2 = new Contact(
  { lastname: 'Bond', firstname: 'James', age: '70' });
const contact3 = new Contact(
  { lastname: 'Hulk' });
```

Indizierte Interfaces

```
interface StringArray {
  [index: number]: string;
}

let authors: StringArray = ['Hoppe', 'Huck', 'Hall'];
```

[1] In funktionalen Programmiersprachen bekanntes Konzept nach dem Gedicht von James Whitecomb Riley

Interfaces für Funktionen

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
const searchFunc: SearchFunc =
  function(source: string, subString: string): boolean {
    return source.search(subString) > -1;
  };
```

oder schneller mit *Arrow-Funktionen* (siehe unten)

```
const searchFunc: SearchFunc =
  (source, subString) => source.search(subString) > -1;
console.log(searchFunc('Hello', 'el'));
```

Interfaces für Klassen

```
interface SwimmerInterface {
  weight: number;
  swim(meters: number): boolean;
}
class Swimmer extends Athlete implements SwimmerInterface {
  constructor (name: string, city: string, public weight: number,
    fitnesslevel?: number) {
    super(name, city, fitnesslevel);
  }
  swim(meters: number): boolean { … }
}
```

optionale Parameter nur am Ende einfügbar

Klasse muss alle Eigenschaften und Methoden des Interfaces implementieren

Arrow-Funktionen bzw. Lambda-Ausdrücke

Sind eine Kurzschreibweise für normale Funktionen:

`function() { }`	➔ `() => { }`	
`function(n) { }`	➔ `n => { }`	bei nur einen Parameter
`function(n) { return n + 1; }`	➔ `n => n + 1`	bei nur einen Befehl, {} und return entfällt

```
const numbers: Array<number> = [0, 1, 2, 3, 4];
```

```
const even = numbers.filter(
  function(value:number):boolean{
    return value % 2;
  }
);
```

➔

```
const even = numbers.filter(
  value => value % 2);
```

liefert `[1, 3]`

Beispiel

```
class Person {
  constructor(
    public name: string,
    public birthdate: Date,
    public height: number,
    public male: boolean,
    public emailaddresses: string[],
    public cars: Array<Car>,
    public address: Address,
    public homepage: string,
    public nickname?: string,
    public imagelink?: string
  ) { }

  get ageInYears(): number {
    return Math.trunc((new Date().valueOf() -
      this.birthdate.valueOf()) / (365 * 60 * 60 * 24 * 1000));
  }

  carsYoungerThan(registrationyear: number): Array<Car> {
    return this.cars.filter(
      value => value.registrationyear > registrationyear);
  }
}
```

```
class Car {
  constructor(
    public licenseplate: string,
    public description: string,
    public registrationyear: number,
    public imagelink?: string
  ) { }
}
```

```
class Address {
  constructor(
    public streetnumber: string,
    public postcodetown: string
  ) { }
}
```

```
let persons: Array<Person> = [
  new Person(
    'Sepp Hintner',
    new Date('December 1, 2001'),
    1.87,
    true, [
      'sh@rolmail.net', 'sh@hotmail.com',
      'sh@gmail.com', 'sh@yahoo.com'
    ], [
      new Car('AG670XL', 'Opel Corsa', 1995, 'https://…'),
      new Car('BK368SR', 'Opel Astra', 2000, 'https://…'),
      new Car('EN734SO', 'Opel Insignia', 2017, 'https://…')
    ],
    new Address('Drususallee 33', '39100 Bozen'),
    'http://www.google.de',
    undefined,
    'https://…'),
  new Person( … )
];
```

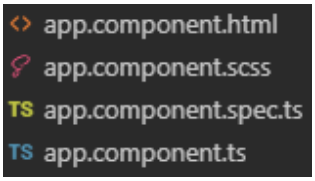
oder

```
let persons: Person[] = [];                          // Leeres Array

persons.push(new Person(…));
persons.push(new Person(…));
console.log(persons[0].ageInYears);
```

Komponenten

- Anwendung besteht aus vielen verschiedenen Komponenten
- Komponente realisiert einen kleinen Teil der Anwendung
- Alle Komponenten sind hierarchisch der *Root-Komponente* (AppComponent) untergeordnet
- Der *Decorator* der Komponente enthält die Konfiguration
- Über den *CSS-Selektor* der Komponente wird diese in eine andere eingebunden
- Komponente besitzt einen Anzeigebereich (View) in dem *Template* dargestellt wird
- Komponente kann individuelle *Stylesheets* beinhalten
- Die *TypeScript-Klasse* der Komponente enthält die Logik

Beispiel Root-Komponente AppComponent



app.component.ts

```
...
@Component({
  selector: 'ta-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title: string = 'Es funktioniert!!!';
}
```

app.component.html

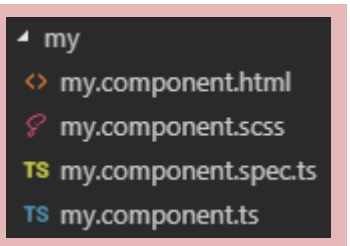
```
<p>
  {{ title }}
</p>
```

{{ Interpolation }}

Komponentenklasse (Logik) *Template* (Darstellung)

```
$ ng generate component my
```

Inhalt der Datei `my.component.ts`:



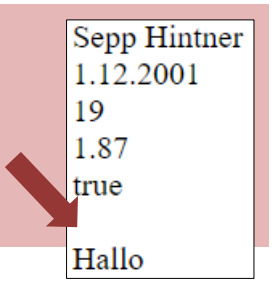

```
@Component({
  selector: 'ta-my',
  templateUrl: './my.component.html',
  styles: [
    '.firstClass { font-weight: bold }'[2],
    '.lastClass { font-style: italic }',
    '.evenClass { color: red }',
    '.oddClass { color: green }'
  ]
})
export class MyComponent implements OnInit {
  person!: Person;
  registrationyear = 1990;

  constructor() { }

  ngOnInit() {
    this.person = new Person('Sepp Hintner', …);
  }

  myClickHandler(value: string) {
    console.log(`Knopf gedrückt ${value}`);
  }
}
```

- Ohne `!` müsste Person spätestens im Konstruktor instanziiert werden
- `ngOnInit()` wird erst gestartet wenn Komponente vollständig initialisiert ist
- Komponente wird in Root-Komponente `app.component.html` durch Selektor `ta-my` eingebunden
- Komponente muss in `AppModule` unter `declarations` registriert werden (wird automatisch von Angular CLI gemacht)
- Zugriff auf Membervariablen muss unbedingt mit `this` erfolgen

[2] Styles werden hinten benötigt

Inhalte der Template-Datei `my.component.html`:

```
{{ person.name }}<br>
{{ person.birthdate.toLocaleDateString() }}<br>
{{ person.ageInYears }}<br>
{{ person.height }}<br>
{{ person.male }}<br>
{{ person.nickname }}<br>
Hallo
```



{{ Interpolation }}

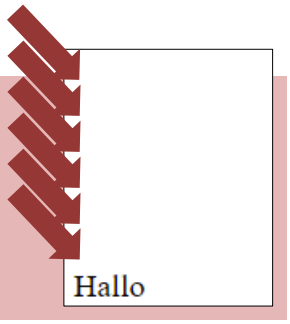
Wert `null` oder `undefined` wird als leerer String ausgegeben

PROBLEM: Wäre `person` noch nicht instanziiert bzw. nicht zugewiesen, würden obige Zeilen Fehler liefern (`TypeError: Cannot read property 'name' of undefined`)

LÖSUNG ***Safe-Navigation-Operator*** **?**

prüft ob **Objekt** vorhanden ist. Wenn nicht wird leerer String ausgegeben

```
{{ person?.name }}<br>
{{ person?.birthdate?.toLocaleDateString() }}<br>
{{ person?.ageInYears }}<br>
{{ person?.height }}<br>
{{ person?.male }}<br>
{{ person?.nickname }}<br>
Hallo
```



! Variableninitialisierung wird dem Programmierer überlassen

? Referenzierung erfolgt nur dann, wenn Variable Inhalt hat

WICHTIG: Ändern die Membervariablen in der Komponente ihre Werte, wird Anzeige sofort aktualisiert!!!

[Property Bindings] und (Event Binding)


```
<a [href]="person.homepage">{{ person.homepage }}</a><br>
<a [href]="'mailto:' + person.emailaddresses[0]">
  {{ person.emailaddresses[0] }}
</a>
```

http://www.google.de
sh@rolmail.net

oder…

```
<a href="{{ person.homepage }}">{{ person.homepage }}</a><br>
<a href="mailto:{{ person.emailaddresses[0] }}">
  {{ person.emailaddresses[0] }}
</a>
```

Ausdruck wird ausgewertet und der Eigenschaft übermittelt

WICHTIG: Property Bindings werden ebenfalls bei Änderungen automatisch aktualisiert

HINWEIS: `[()]` *Two-Way Bindings* (siehe hinten)

#Elementreferenzen

```
<input #id type="text" value="Angular"><br>
{{ id.value }}
<button (click)= "myClickHandler(id.value)">Drück mich</button>
```

Angular1
Angular1 Drück mich

HTML-Element wird mit Namen versehen, durch welchen auf Element und alle seine Eigenschaften zugegriffen werden kann

Aktualisierung erfolgt aber erst wenn ein Ereignis ausgelöst wird

Über Elementreferenz und Event-Binding können Formulareingaben in Komponentenklasse verarbeitet werden

*Strukturdirektiven

Beeinflussen die Struktur des DOMs:

`*ngIf` `*ngFor(let … of …)` `[ngSwitch]`

```
<div *ngIf="person.height > 1.85">Die Person ist groß</div>
<div *ngIf="person.male; else female_content">
  Männlich
</div>
<ng-template #female_content>
  Weiblich
</ng-template>
<ul *ngIf="person.emailaddresses.length">
  <li *ngFor="let emailaddress of person.emailaddresses">
    {{ emailaddress }}
  </li>
</ul>
<div *ngFor="let emailaddress of person.emailaddresses;
  let i=index; let f=first; let l=last; let e=even; let o=odd"
  [class.firstClass]="f" [class.lastClass]="l"
  [class.evenClass]="e" [class.oddClass]="o">
  {{ i + 1 }}. {{ emailaddress }}
  <span *ngIf="f">Erstes</span>
  <span *ngIf="l">Letztes</span>
</div>
<div [style.font-size.px]="person.ageInYears * 2">
  {{ person.ageInYears }}
</div>
<div [ngSwitch]="person.address.postcodetown">
  <span *ngSwitchCase="'39100 Bozen'">Kommt von Bozen</span>
  <span *ngSwitchCase="'39012 Meran'">Kommt von Meran</span>
  <span *ngSwitchDefault>Kommt von anderswo</span>
</div>
```

- Tag wird nicht gerendert, wenn Bedingung in `*ngIf` `false` ergibt
- `[class.`*Klassenname*`]="`*Ausdruck*`"`
  Klasse wird einem Element zugewiesen, wenn Ausdruck wahr ist
- `[style.`*Stilname*`.`*Einheit*`]="`*Ausdruck*`"`
  Ein Stil wird einem Element zugewiesen und zwar auf den Wert den Ausdruck liefert. Einheit ist optional

Die Person ist groß
Männlich

- sh@rolmail.net
- sh@hotmail.com
- sh@gmail.com
- sh@yahoo.com

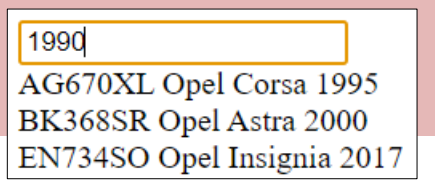
1. sh@rolmail.net Erstes
2. sh@hotmail.com
3. sh@gmail.com
4. sh@yahoo.com Letztes

19

Kommt von Bozen

Einfache Formularverarbeitung

```
<input type="text" [(ngModel)]="registrationyear">
<div
  *ngFor="let car of person.carsYoungerThan(registrationyear)">
  {{ car.licenseplate }}
  {{ car.description }}
  {{ car.registrationyear }}
</div>
```

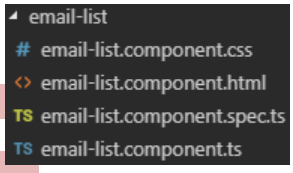


- In `app.module.ts` unter `imports` `FormsModule` importieren
- `ngModel` bindet Membervariable der Komponente an ein Eingabefeld
- In Komponentenklasse muss Membervariable `registrationyear` vorhanden sein. Diese wird synchronisiert (siehe Datei `my.component.ts`)

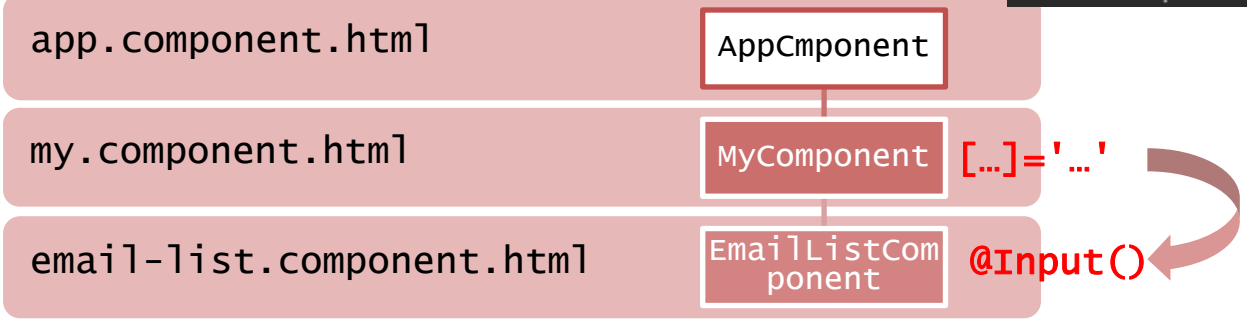
`[()]` ***Two-Way Bindings:*** Werte werden zw. Komponente und Template sofort synchronisiert. Reihenfolge wichtig: *Banana Box*

Komponentenhierarchien und Werteübergabe

Einzelne Teile der Anwendung in Komponenten auslagern, die einzeln wartbar, wiederverwendbar und testbar sind




```
$ ng generate component email-list
```



Integration in my.component.html und Wertübergabe

```
…
<ta-email-list [emailaddresses]="person.emailaddresses">
</ta-email-list>
```

Übergabe der Daten über *Property Binding*

Werte in Komponente auslesen email-list.component.html

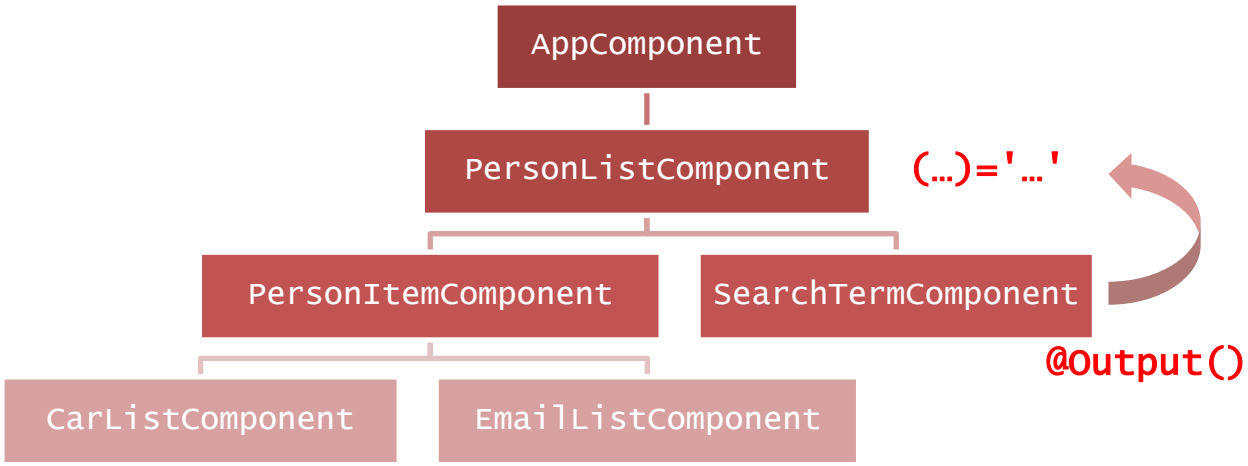
```
@Component({
  selector: 'ta-email-list',
  templateUrl: './email-list.component.html',
  styles: []
})
export class EmailListComponent implements OnInit {
  @Input() emailaddresses!: Array<string>;

  …
  ngOnInit() {
    console.log(this.emailaddresses);
  }
}
```

- `ngOnInit()` muss verwendet werden, denn im Konstruktor ist *Input-Binding* noch nicht initialisiert
- `ngOnChange()` würde bei jeder Änderung des Input-Bindings aufgerufen

Template email-list.component.html

```
<h2>E-Mail-Adressen</h2>
<ul *ngIf="emailaddresses?.length; else empty_list_content">
  <li *ngFor="let emailaddress of emailaddresses">
    <a href="mailto:{{emailaddress}}">{{ emailaddress }}</a>
  </li>
</ul>
<ng-template #empty_list_content>
  Keine E-Mail-Adressen festgelegt
</ng-template>
```

Werte von der Komponente SearchTermComponent zurückliefern




```
export class SearchTermComponent implements OnInit {
  @Output('searchTerm') changeEvent: EventEmitter<string>;
  constructor() {
    this.changeEvent = new EventEmitter<string>();
  }
  …
  emitSearchTermChange(searchTerm: string) {
    this.changeEvent.emit(searchTerm);
  }
}
```

```
<input #searchTerm type="text"
  (keyup)="emitSearchTermChange(searchTerm.value)">
```

Auf Änderungen in `PersonListComponent` reagieren

```
<ta-search-term (searchTerm)="onSearchTermChange($event[3])">
</ta-search-term>
```

```
onSearchTermChange(searchTerm: string) {
  console.log(searchTerm);
}
```

`EventEmitter` basiert auf der `Observable`-Klasse (siehe nächstes Kapitel)

[3] Konvention: Asynchrone Objekte werden mit $ bezeichnet

Responsives Web-Design mit Angular Material und Flex-Layout

Angular Material

Offizielle Implementierung von *Google Material Design* für Angular

- *Material Design* ist Designsprache zur Gestaltung von App-/Web-Oberflächen
- Enthält eine Vielzahl von Komponenten (für Formulare, Navigation, Layouts, Knöpfe, Dialoge, Tabellen, usw.)
- Komponenten sind barrierefrei
- Theaming-Support

`https://material.angular.io`

Integration in ein bestehendes Projekt

WICHTIG: Beim Anlegen des Projektes muss SCSS als Beschreibung der Styles eingestellt worden sein

```
$ ng add @angular/material
```

Dabei muss

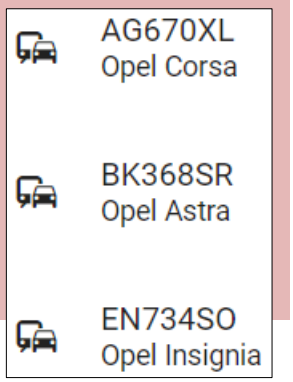
- *Theme* ausgewählt,
- *Global Angular Material typography styles* nicht gesetzt[4]
- und *BrowserAnimationsModule* integriert werden (stellt Animationen dar)

[4] Würden diese gesetzt, so würden auf alle HTML-Tags die Material-Styles angewendet – z.B. auch auf <h1>

Arbeiten mit Angular Material-Komponenten

1. Schritt: Modul importieren (*)

```
import { MatListModule } from '@angular/material/list';
import { MatIconModule } from '@angular/material/icon';
…
@NgModule({
  …
  imports: [
    MatListModule,
    MatIconModule
  ],
})
export class AppModule { }
```



2. Schritt: Komponente verwenden

```
<mat-list *ngIf="person.cars && person.cars.length">
  <mat-list-item *ngFor="let car of person.cars">
    <mat-icon matListIcon>commute</mat-icon>
    <h3 matLine>{{ car.licenseplate }}</h3>
    <p matLine>{{ car.description }}</p>
  </mat-list-item>
</mat-list>
```

Flex-Layout

Packet zum responsiven Anordnen und Ausblenden von Elementen

`https://github.com/angular/flex-layout`

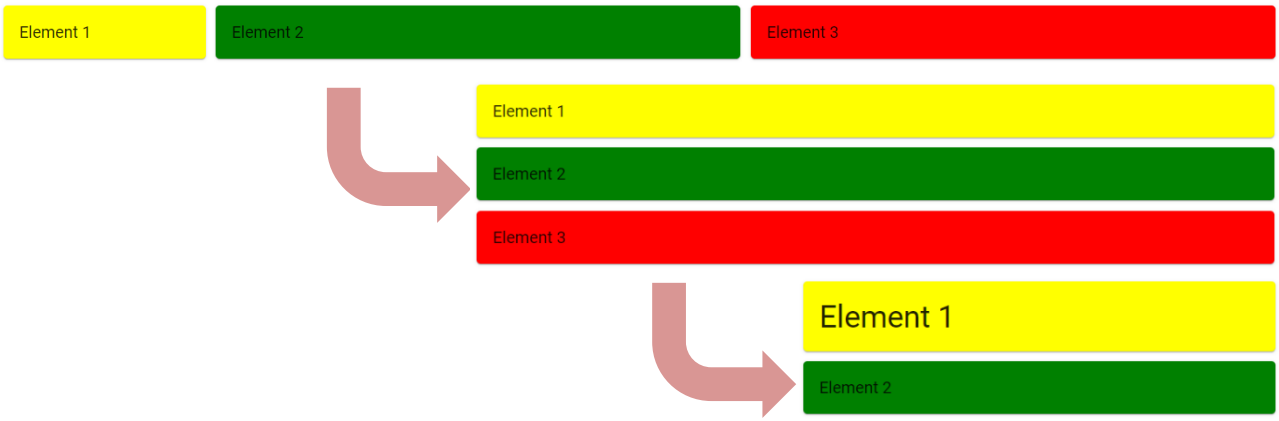
```
$ npm install @angular/flex-layout
```

Erstellen eines Responsiven Layouts

Breakpoints

xs	[0, 599px]	`lt-sm`	
sm	[600px, 959px]	`lt-md`	`gt-xs`
md	[960px, 1279px]	`lt-lg`	`gt-sm`
lg	[1280px, 1919px]	`lt-xl`	`gt-md`
xl	[1920px, 5000px]		`gt-lg`

- Zwischen Elementen sollen 10px Abstand sein (`fxLayoutGap`),
- Elemente sollen sich Anzeigebereich teilen (`fxFlex`),
- auf großen Bildschirmen (`gt-md`) nebeneinander angezeigt werden, aber dabei *Element 1* nur 200px breit sein,
- auf kleinen (`lt-md`) untereinander und
- auf sehr kleinen Bildschirmen (`lt-sm`) soll letztes Element verschwinden und Schrift des ersten Elements (`ngStyle.xs`) größer werden



1. Schritt: Modul importieren wie vorher (*)

```
import { FlexLayoutModule } from '@angular/flex-layout';
…
```

2. Schritt: Integration ins Template

Auch ngClass verwendbar

```
<div fxLayoutGap="10px" fxLayout.gt-md="row"
  fxLayout.lt-lg="column" >            <!-- Zeile eigentlich unnötig -->
  <mat-card fxFlex.gt-md="200px"
    [ngStyle.xs]='{"font-size.px": 30}'>
    Element 1
  </mat-card>
  <mat-card fxFlex>Element 2</mat-card>
  <mat-card fxFlex fxHide.lt-sm>Element 3</mat-card>
</div>
```