

Angular



Vorstellung

- Ihr Trainer:

Vadzim Naumchyk

vadzimn@ppedv.de

- Kurse bei ppedv:
 - JavaScript
 - HTML und CSS
 - Responsive Design
 - Angular



Die ppedv AG

- Firmensitz in Burghausen (Bayern)
- Büros in verschiedenen Städten:
- Ca. 50 Mitarbeiter
- Schulungen für nahezu alle Microsoft-Technologien
- Konferenzen, Camps, Blog



Agenda

- Einführung in Angular und erste Komponenten
- Einführung in ES2015+ und TypeScript
- Komponenten – Templates, Properties und Events
- Forms
- Daten-Services
- Server-Kommunikation
- Routing
- Material Design - Komponenten

Einführung in Angular

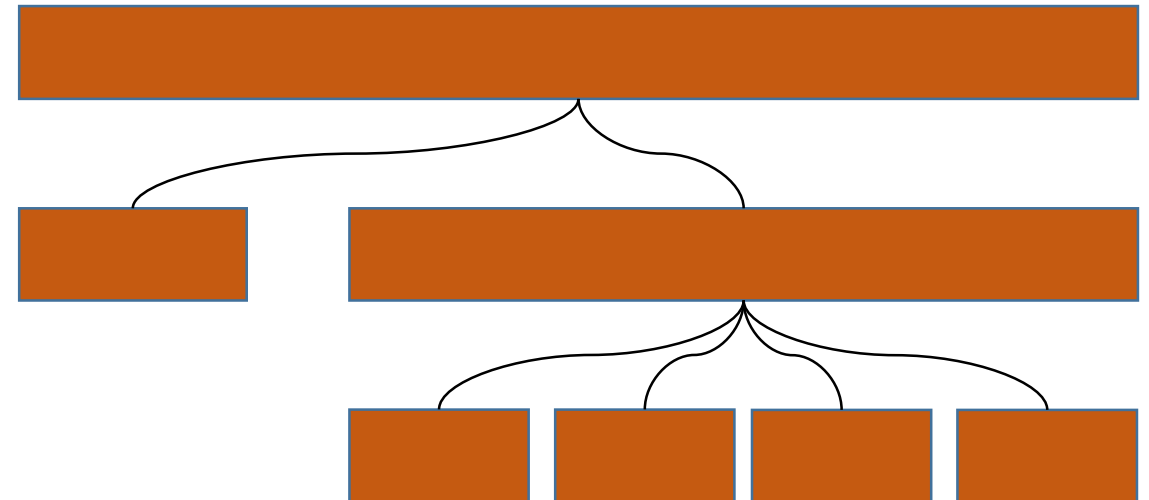
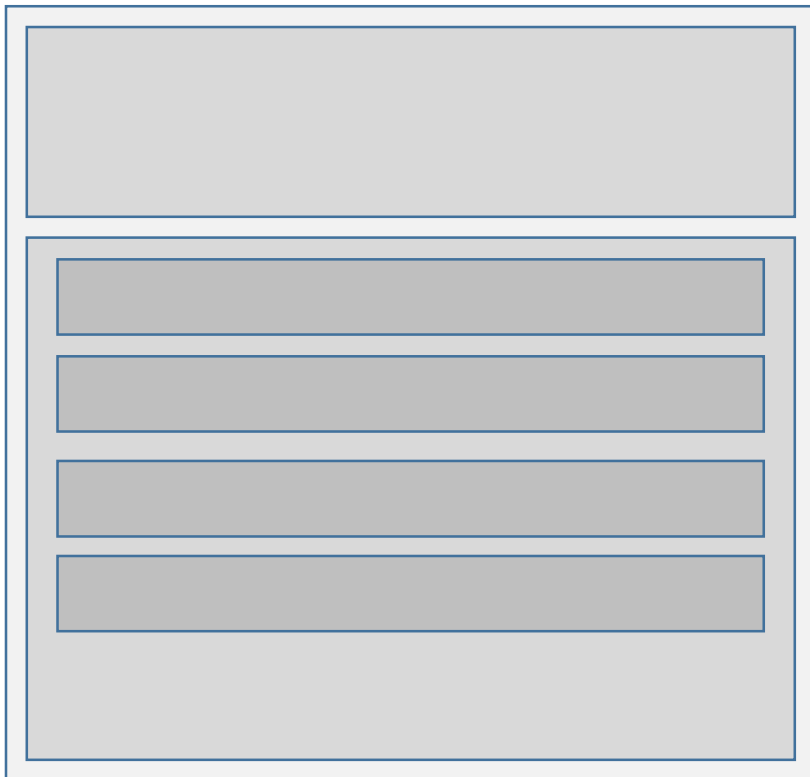
Eine kurze Geschichte von Angular

- AngularJS: Entwicklungsbeginn 2009
- Angular 2: September 2016 – Komplette Neuentwicklung
- Angular 4: März 2017
- Angular 5: November 2017
- Angular 6: Mai 2018
- Angular 7: Oktober 2018

Architektur moderner SPAs

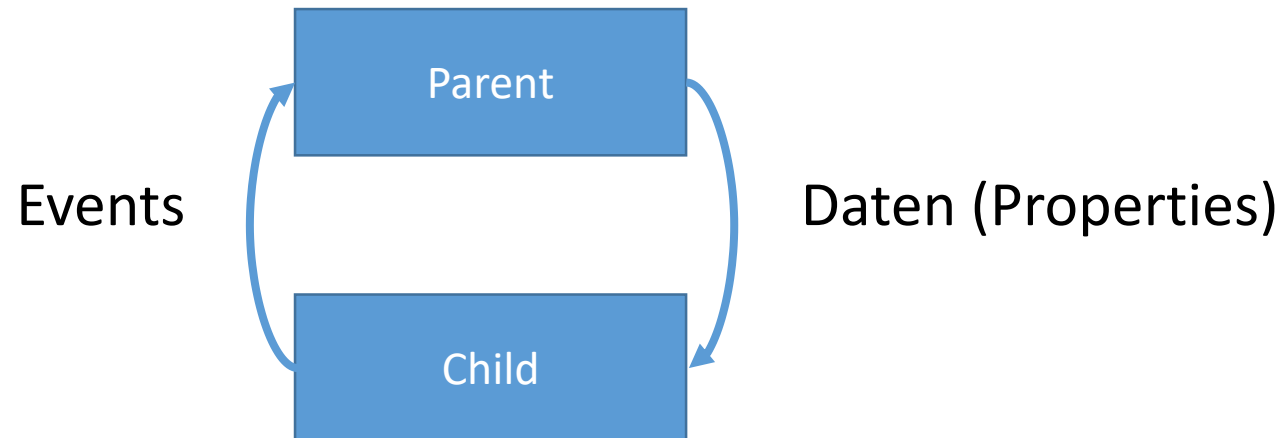
- Modernes JavaScript / TypeScript
 - Module
 - Klassen
- Deklarativ / Datengetrieben
 - Man ändert das Model, das View wird automatisch aktualisiert
- Komponenten – Struktur
 - „eigene“ HTML-Tags
 - Properties und Events
 - üblicherweise unidirektionaler Datenfluss (vom Eltern- zum Kindelement)

Komponentenstruktur



Datenfluss

- Moderne Architektur (Angular 2+, React, Vue): Daten fließen vom Eltern- zum Kindelement
- Angular: Einfache Syntax zur bidirektionalen Datenbindung in Sonderfällen



Komponentenstruktur: Tags und Properties

- Beispiel:

```
<todo-item [title]=" 'groceries' " [completed]="false">  
</todo-item>
```

Komponenten

Was ist eine Komponente?

- Möglichkeit, eigene HTML-Tags zu definieren
- Komponentendefinition beinhaltet:
 - HTML-Template
 - CSS-Stil
 - Properties
 - Events

Erstes Beispiel: Erste App & Komponenten

Wir lernen:

- Angular CLI zum Erstellen von Projekten/Komponenten/...
- Struktur einer Komponentendefinition
- Einbinden von Komponenten
- Komponenten - Template
- Stil-Zuweisung zu Komponenten

Erstes Beispiel: Angular CLI

- Wir erstellen ein neues Projekt mittels:
 - `ng new firstproject`
 - Das dauert ... im Hintergrund werden npm-Abhängigkeiten installiert und im Ordner *node_modules* abgelegt
- Wir lassen das Projekt auf einem lokalen Server ausführen:
 - `cd firstproject`
 - `ng serve`
- Wir sehen das laufende Projekt auf <http://localhost:4200>

Erstes Beispiel: Angular CLI

Überblick über Angular CLI:

- `ng new $projectname`: Erstellt neues Angular-Projekt
- `ng serve`: Startet den Testserver
- `ng generate component $name`: Erstellt eine neue Komponente
- `ng generate service $name`: Erstellt ein neues Service
- `ng build --prod`: Führt einen Production-Build aus (im dist-Ordner)

Entwicklung mit node.js und npm

- node.js: JavaScript-Runtime
 - Angular CLI, Angular Testserver
 - Unit Tests
- npm: Paketmanager für node.js
 - Zur Verwaltung von Abhängigkeiten
 - Pakete landen im Ordner *node_modules*
 - Konfiguration in *package.json*

Erstes Beispiel: Projektstruktur

- Angular CLI erstellt umfangreiche Projektstruktur
- Uns interessiert hauptsächlich der Ordner *src/app*
- Andere Ordner / Dateien:
 - **package.json**: npm-Konfiguration
 - **karma.conf, protractor.conf**: Tests
 - **tsconfig.json**: Typescript-Konfiguration
 - **src/index.html**: Einstiegspunkt
 - **src/polyfills.ts**: Polyfills für „ältere“ Browser
 - **src/app/**: eigentliche Angular-App

src/polyfills.ts

- Wird für volle Unterstützung älterer Browser benötigt (z.B. IE 9-11)
- Zum testen auf / deployen für ältere Browser: entsprechende Zeilen „einkommentieren“ und entsprechende Abhängigkeiten mittels npm installieren
- Details: <https://angular.io/guide/browser-support>

Erstes Beispiel: Komponentenstruktur

- Mit angular-cli (ng) erstellte Komponenten gliedern sich in drei Dateien, zB:
 - app.component.html (Template)
 - app.component.css (auf Komponente beschränkter Stil)
 - app.component.ts (Programmcode)

Erstes Beispiel: Änderungen an der Komponente

- Wir ändern unsere eben erstellte Komponente etwas ab:
 - Die Schriftfarbe soll rot sein
 - Die Komponente soll „Hallo, \$city“ ausgeben, wobei der Name in der .ts-Datei definiert wird
- Wie können wir Variablen in unser HTML einsetzen?
 - Wir definieren die Variablen als Properties unserer Komponente
 - Im HTML können wir sie mittels {{ Variablenname }} einbinden

Erstes Beispiel: Änderungen der Komponente

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  city = 'Rostock';
}
```

Erstes Beispiel: Änderungen der Komponente

```
<!-- app.component.html -->
```

```
<h1>Hallo, {{city}}!</h1>
```

Erstes Beispiel: weitere Komponenten

Wir erstellen folgende weitere Komponenten:

- `<app-time>` - Komponente, die die aktuelle Uhrzeit anzeigt
- `<app-dice>` - Komponente, die eine Zufallszahl von 1 bis 6 zeigt

Erstes Beispiel: Weitere Komponenten

Befehl zum Erstellen neuer Komponenten:

```
ng generate component time
```


Erstes Beispiel: Time-Komponente

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

@Component({
  selector: 'app-time',
  templateUrl: './time.component.html',
  styleUrls: ['./time.component.css']
})
export class TimeComponent implements OnInit {
  // wir deklarieren eine Time-Variable vom Typ "Date"
  time: Date;

  constructor() {
    this.time = new Date();
  }
  ngOnInit() {
  }
}
```

Erstes Beispiel: Dice-Komponente

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

@Component({
  selector: 'app-dice',
  templateUrl: './dice.component.html',
  styleUrls: ['./dice.component.css']
})
export class DiceComponent implements OnInit {
  // wir deklarieren eine number-Variable vom Typ "number"
  number: number;

  constructor() {
    this.number = Math.ceil(Math.random()*6);
  }
  ngOnInit() {
  }
}
```

Beispiel ausführen

\|/ - templateUrl vs. template

- anstatt von templateUrl kann template verwendet werden:

```
templateUrl: './todos.component.html'
```

oder:

```
template:
```

```
`
```

```
<p>todos works!</p>
```

```
<h2>Todo {{todo.title | uppercase}}</h2>
```

```
<div>id: {{todo.id}}</div>
```

```
<div>erledigt: {{todo.done}}</div>
```

```
`
```

- Datei todos.component.html hat also den Inhalt, der in template steht.

\|/ - ngOnInit vs. constructor()

```
export class App implements OnInit {
```

```
    constructor(){
```

```
        //wird aufgerufen vor ngOnInit()
```

```
    }
```

```
    ngOnInit(){
```

```
        //wird aufgerufen nach dem Konstruktor und nach
```

```
    ngOnChanges()
```

```
    }
```

```
}
```

\|/ - lifecycle sequence

After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

- ngOnChanges()
- ngOnInit()
- ngDoCheck()
- ngAfterContentInit()
- ngAfterContentChecked()
- ngAfterViewInit()
- ngAfterViewChecked()
- ngOnDestroy()

ES2015+ und TypeScript

TypeScript

- Typ Annotationen
- Dekoratoren
- public & private Fields
- ...

ES2015+

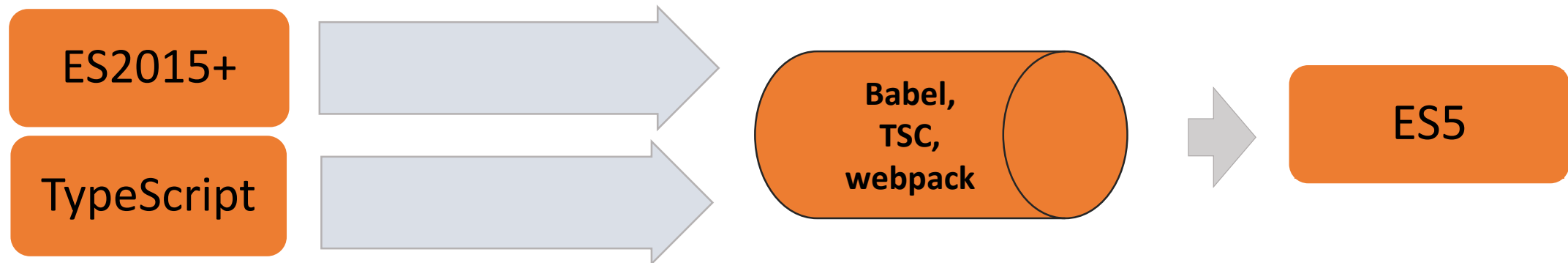
- Module
- Klassen
- Arrow functions (Lambda)
- ...

ES5

- Status Quo

Kompilierung

- Aktuelles JavaScript (von Browsern unterstützt): ES5 (EcmaScript 5)
- ES2015 (ES6), ES2016, ES2017: neue Funktionen
- TypeScript: Erweiterung von ES



Neuerungen in ES2015+

Module & Imports

- Möglichkeit, Funktionalität aus anderen js-Dateien zu importieren – kein globaler Namespace mehr
- Benötigt einen Bundler, z.B. webpack

```
// user.js
export class User {
  ...
}
```

```
// main.js
import { User } from 'user.js';
```

let

- Neue Alternative zu *var* – mit leicht anderem Scoping
- Scope: umgebende geschwungene Klammern
- Empfehlung: verwenden anstatt von *var* wo möglich

```
let a = 3;
```

Arrow-Funktion / Lambda

- Kurzschreibweise für anonyme Funktionen
- Erfasst ***this*** aus dem umgebenden Kontext

```
let multiply = (a,b) => { return a * b };
```

```
let multiply = (a,b) => a * b;
```

Klassen

- Ersetzen die alten Konstrukturfunktionen und Prototypen

Klassen

```
class Person {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
    hello() {  
        return `My name is ${this.firstName} ${this.lastName}`;  
    }  
}
```

Array - Iteration

Mit der Syntax „for (let item of array) {...}“ können wir über die Einträge in einem Array iterieren

```
let names = ['Anna', 'Bernhard', 'Caro'];
```

```
for (let name of names) {  
    console.log(name);  
}
```


Template – Strings

- Neue Syntax zum *Erstellen* von Strings
- Werden mit Backticks (`) begrenzt
- Erlauben mehrzeilige Strings und Interpolation:

```
let name = "Vadzim";  
let greeting = `Hallo, ${name}!  
                Willkommen bei ES2015!`;
```

TypeScript

TypeScript

- Obermenge von JavaScript mit Erweiterungen:
 - **Static Typing**
 - Decorators
 - Public / Private Properties

Static Typing

- Datentypen *können* angegeben werden und unterstützen insbesondere die Entwicklungsumgebung:
 - Auto-Vervollständigung
 - Fehlermeldungen bei nicht passenden Datentypen
- Beim build: TypeScript wird in JavaScript übersetzt, alle Typeninformationen gehen dabei verloren

Typsystem in TypeScript: Variablen

Beispiele:

```
let age: number = 32;  
let name: string = 'Andreas';
```

Typsystem in TypeScript: Funktionen

Wir können Parametertypen und Rückgabetypen angeben

```
function repeatString(text: string, times: number): string {  
    return ...;  
}
```

Typsystem in TypeScript: Funktionen

Optionale Parameter:

```
function buildName(  
    firstName: string, lastName?: string): string {  
    return firstName + ' ' + lastName;  
}
```

Typsystem in TypeScript: Arrays

Beispiele:

```
let names: string[] = ['Anna', 'Bernhard', 'Caro'];
```

```
let amounts: number[] = [3, 10, 23];
```

// Alternative Syntax

```
let names: Array<string> = ['Anna', 'Bernhard', 'Caro'];
```


Typsystem in TypeScript: Objekte

Beispiele:

```
let p: {name: string, age: number} = getPerson();  
console.log(p.age);
```

// oder

```
interface Person {  
  name: string,  
  age: number  
}
```

```
let p: Person = getPerson();
```

Typsystem in TypeScript: Void

void: umfasst *undefined* und *null*

```
function warnUser(): void {  
    alert('this is a warning message');  
}
```

Typsystem in TypeScript: Any

any: lässt alle Typen zu

```
let inputBox: any = document.querySelector('#inputbox');
```

Decorators

Mit Decorators lassen sich Funktionen und Klassen nach ihrer Erstellung mittels einer Funktion – dem Decorator – verändern

Beispiel:

```
// Hypothetischer cache-Decorator,  
// der die Resultate eines Funktionsaufrufs speichert  
import { cache } from 'cache.js';  
  
@cache  
function getResults() {  
    return this.results;  
}
```

Decorators

In Angular werden Decorators verwendet, um Metadaten zu einer Klasse zu ergänzen:

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  name = 'Anton';  
}
```

Private & Public Properties

```
class ClockComponent {  
    private formatTime(time) {  
        return ...  
    }  
    public start() {  
        ...  
    }  
}
```

Private & Public Properties im Constructor

```
class Person {  
    constructor(public name: string, public age: number) {}  
}
```

Kurzform für:

```
class Person {  
    name: string;  
    age: number;  
    constructor(name: string, age: number) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

Properties

Properties & Events

Properties und Events stellen die wichtigsten Mechanismen dar, mit denen Komponenten in SPAs miteinander interagieren.

In Angular verwendet man auch die Begriffe:

input = Property

output = Event

Properties

Bisher haben wir Inhalte von Komponenten entweder fix vorgegeben oder z. B. durch Zufallszahlen generieren lassen.

Richtig interessant werden Komponenten erst, wenn wir sie durch Übergabe von Parametern anpassen können.

Properties

Mittels Properties (in Angular-Sprechweise: Inputs) können Werte von Elternkomponenten an eine Kindkomponente übergeben werden.

Das funktioniert sowohl für bestehende Tags (wie z. B. src bei img) als auch für neu definierte:

```
<img [src]="getImageUrl()">
```

```
<app-rating [stars]="4"></app-rating>
```

Properties: Beispiel Ratingkomponente

Wir erstellen eine Komponente, die eine bestimmte Anzahl an Sternen anzeigt, die sich über ein HTML-Attribut steuern lässt:

```
<app-rating [stars]="4"></app-rating>
```



Properties: Beispiel Ratingkomponente

Zum Üben erstellen wir eine neue Angular-App:

```
ng new fundamentals
```

(wir achten dabei darauf, dass wir im Kursordner sind)

...und wir erstellen eine neue Komponente:

```
ng generate component rating
```

Properties: Beispiel Ratingkomponente

Wir verändern rating.component.ts folgendermaßen:

- Zunächst importieren wir Input:
 - `import { Input } from '@angular/core';`
- Weiters setzen wir Typinformationen fest:
 - `@Input() stars: number;`
 - `starString: string;`
- In `ngOnInit()` erstellen wir den Wert `starString`:
 - `this.starString = '*'.repeat(this.stars);`

Properties: Beispiel Ratingkomponente

Unser Template sieht folgendermaßen aus:

```
<div>  
    {{ starString }}  
</div>
```

Properties: Beispiel Ratingkomponente

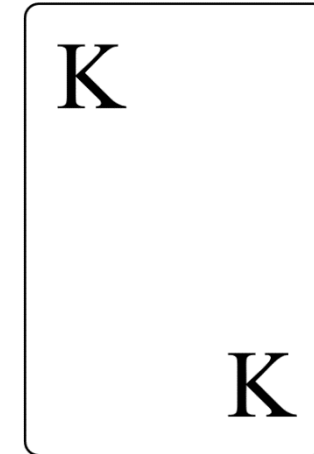
Und unser CSS:

```
div {  
    font-family: monospace;  
    font-size: 40px;  
    color: gold;  
}
```


Properties: Weitere Beispiele

III VII IX X

```
<app-roman-number [number]="3"></app-roman-number>  
<app-roman-number [number]="7"></app-roman-number>  
<app-roman-number [number]="9"></app-roman-number>  
<app-roman-number [number]="10"></app-roman-number>
```



```
<app-card [value]="'K'"></app-card>
```



```
<app-diashow  
  [images]="  
    'https://picsum.photos/400/300?image=0',  
    'https://picsum.photos/400/300?image=10',  
    'https://picsum.photos/400/300?image=20',  
    'https://picsum.photos/400/300?image=30' ]">  
</app-diashow>
```

Templates

Angular-Templates im Detail

Bisher: Datenbindung:

- Inhalte: {{ }}
- Attribute: []

Weitere Features:

- *ngFor
- *ngIf
- style
- class
- pipes
- #templateReferenceVariables

Angular-Templates im Detail: *ngFor

Mit *ngFor können wir HTML-Elemente wiederholen.

Beispiel:

```
<div *ngFor="let todo of todos">  
  {{ todo.text }}  
</div>
```

*ngFor wiederholt das Element, auf das es angewendet wird (hier: das div-Element)

Angular-Templates im Detail: *ngFor

Optional können wir mit Hilfe der *index*-Variable mitzählen:

```
<div *ngFor="let todo of todos; index as i">  
  {{i}}: {{ todo.text }}  
</div>
```

Angular-Templates im Detail: *ngFor

Übung:

Wir zeigen die römischen Zahlen aus dem Array [1, 5, 10] mittels ngFor und der zuvor definierten Komponente

Angular-Templates im Detail: *ngIf

Mit *ngIf können wir ein Element unter bestimmten Bedingungen ein- oder ausblenden.

Wir verwenden es beispielsweise so:

```
<div *ngIf="hasErrors()">We encountered an Error</div>
```

Oder so:

```
<div *ngIf="item.importance >= 3">{{ item.text }}</div>
```

Angular-Templates im Detail: Stile

Für jede Komponente können wir CSS-Stile festlegen. Diese betreffen dann nur eine Komponente.

Dazu gibt es zwei Möglichkeiten in der Konfiguration:

```
styleUrls: [ './app.component.css' ]
```

oder

```
styles: [ `h1 {..} ....` ]
```


Angular-Templates im Detail: Stile

In Angular-Stildefinitionen gibt es einen besonderen Selektor:

```
:host {  
  display: block;  
}
```

Der Host-Selektor bezieht sich auf das Komponententag selbst.

Angular-Templates im Detail: Stile

Für einzelne HTML-Elemente können wir direkt mit einer Angular-eigenen Syntax Stile via Properties setzen:

```
<div [style.color]="getTextColor()">...</div>
```

```
<span [style.font-size.px]="getFontSize()">...</span>
```

```
<div [style.width.%]="100 / n">...</div>
```

Angular-Templates im Detail: Stile

Beispiel zu Stilen:

Wir ändern `<app-roman-number>` so ab, dass sich die Größe proportional zum Wert ändert.

Angular-Templates im Detail: Klassen

Auch für Klassen bietet Angular eine Syntax:

```
<div [class.important]="isImportant()">...</div>
```

```
<span [class.negative]="item.amount < 0">  
  {{ item.amount }}  
</span>
```

Templates im Detail: Pipes

- Pipe = Im wesentlichen eine Funktion, die im Template zur string-Formatierung zur Verfügung steht
- Beispiele:
 - Today is `{{ today | date }}`
 - My name is `{{ name | uppercase }}`
 - The price is `{{ price | number:'1.2-2' }}`
 - Debugging information: `{{ todo | json }}`
 - Total amount: `{{ total | currency:'EUR':true:'1.2-2' }}`

Templates im Detail: Template reference variables

Mit Template reference variables können wir auf Elemente aus dem Template zugreifen. Dazu verwenden wir das #-Zeichen.

```
Todo: <input #newtodo>  
<button (click)="addTodo(newtodo.value)">  
  Add  
</button>
```

Template reference variables werden insbesondere bei Form-Events und Form-Validierung eine Rolle spielen.

Events

Standard-Events

Wir können alle Standard-DOM-Events über die folgende Syntax überwachen:

```
<div (eventname)="eventHandler()">...</div>
```

Liste von Standard-DOM-Events:

https://www.w3schools.com/jsref/dom_obj_event.asp

Standard-Events: Beispiele

```
<button (click)="increase()"> + </button>
```

```
<input (keydown)="onKeyDown()">
```

Standard-Events: Event-Objekt

Das JavaScript Event-Objekt können wir über den Parameter mit dem Namen *\$event* erhalten.

```
<input (keydown)="onKeyDown($event)">
```

```
onKeyDown(event: KeyboardEvent) {  
    event.preventDefault();  
    this.key = event.key;  
}
```

Standard-Events: Beispiel

Wir erstellen eine Komponente mit dem Namen `<events-helloworld>`.

Sie enthält einen Button; beim Drücken erscheint ein `alert()` - Fenster mit dem Text „button clicked“.

Standard-Events: weitere Beispiele

- `<app-home-away>`:
 - Komponente, die sich je nachdem ob der Mauszeiger drinnen ist verändert:

Away

Home

- `<app-click-counter>`:
 - Button, der die Klickanzahl mitzählt
- `<app-keycode-display>`:
 - Input-Feld, das zu einem Tastendruck den gedrückten Buchstaben in einem extra `<div>` anzeigt

Standard-Events: weitere Beispiele

fontsize-selector:

```
<input
  type="number"
  (
```

Das input-Event tritt ein, wenn sich der Wert des input-Elements ändert.

\$event.target.value ist der neue Wert.

[MDN: input-Event](#)

Event-„Filter“

```
(keyup.enter)="onEnter()"
```

Mit der obigen Syntax können Events auf bestimmte Kategorien beschränkt werden.

Eigene Events

- Definition eigener Events:
 - Eventname
 - Evtl zugehörigen Wert (zB Zahl, String, oder auch ein komplexeres Objekt) – dieser wird im *\$event*-Parameter übergeben
 - Für den zugehörigen Wert muss ein Typ festgelegt werden (kann auch void sein).

Eigene Events: Definition

```
import {Output, EventEmitter } from '@angular/core';
```

```
[...]
```

```
@Output() tick: EventEmitter<number> = new  
EventEmitter<number>();
```

```
[...]
```

```
this.onTick.emit(this.remaining);
```


Eigene Events: Beispiel

- Timer mit Events: timer-start, timer-tick, timer-over
 - Nutze diese Events für Beispielnachrichten:
 - „Der Timer ist gestartet“
 - „nur noch 3 Sekunden“
 - „Timer abgelaufen“

Beispiel für Verwendung:

```
<app-timer
    [time]="10"
    (start)="onStart()"
    (tick)="onTick($event)">
</app-timer>
```

Eigene Events - Beispiel

Beispiel: rating-output: Ähnlich wie die Rating-Komponente zuvor, nur Datenfluss in die andere Richtung

Die Komponente reagiert auf click-Events und triggert dann selbst ein *ratingChanged*-Event mit der entsprechenden Sternenzahl. Sie kann folgendermaßen verwendet werden:

```
<app-rating-output  
  (ratingChanged)="onRatingChanged($event)">  
</app-rating-output>
```

Beispiel: Todo-App

Todo-App

- *ngFor, *ngIf
- @Input(), @Output, EventEmitter

Inputs & Forms

Inputs & Forms - Grundlagen

Um Forms und Inputs mit Angular nutzen zu können, müssen wir zunächst in `app.module.ts` das *FormsModule* importieren:

```
import {FormsModule} from '@angular/forms';  
  
...  
imports: [  
    BrowserModule,  
    FormsModule  
],
```

Inputs: ngModel

- Mit Hilfe von *ngModel* können wir Änderungen an einem Input überwachen lassen

Inputs: ngModel

Einfaches (theoretisches!) Beispiel mit standard HTML-Attributen und template reference variables:

```
<input ngModel #myInput required minlength="3"><br>  
value: {{ myInput.value }} <br>  
valid: {{ myInput.validity.valid }}
```

Üblicherweise greift man nicht auf das *input*-Element selbst zu, sondern auf dessen *ngModel*-Controller:

```
<input ngModel #myInput="ngModel" required minlength="3"> <br>  
value: {{ myInput.value }} <br>  
valid: {{ myInput.valid }} <br>  
touched: {{ myInput.touched }} <br>  
pristine: {{ myInput.pristine }}
```


Inputs: ngModel

```
<input ngModel #myInput="ngModel" required  
minlength="3">
```

Was passiert hier?

Mit *ngModel* bringen wir Angular dazu, den Inhalt des Inputs zu überwachen.

Mit *#myInput="ngModel"* setzen wir dann eine Variable, die auf das entsprechende Datenmodell verweist. Zu beachten: Der Wert rechts („ngModel“) ist fest vorgegeben, den linken Namen (myInput) können wir selbst bestimmen.

Inputs: ngModel

value: {{ myInput.value }} **
**

valid: {{ myInput.valid }} **
**

touched: {{ myInput.touched }} **
**

pristine: {{ myInput.pristine }}

Folgende Eigenschaften des *ngModel*-Controllers können wir überwachen:

- value: Wert – dieser ist oft automatisch vom passenden Typ (zB bei type="number" oder type="checkbox" – nicht aber bei type="date")
- valid
- touched: ändert sich auf true, wenn der Fokus in das Feld gesetzt wird und dann wieder auf etwas anderes
- pristine: ändert sich auf false, sobald der Wert zum ersten Mal geändert wird.

Beispiel

- app form-exercises
 - component example1

Übung zu ngModel: Passwort

Wir setzen eine Passworteingabe um:

- Es soll zwei Eingabefelder geben, deren Inhalt wir mit *ngModel* überwachen.
- Solange die Eingabefelder unterschiedliche Werte haben, soll der zugehörige *OK*-Button auf *disabled* gesetzt sein und darunter (in einem extra `<div>`) eine Warnung angezeigt werden.

ngModel und two-way data binding

- Bisher haben wir ngModel nur im Template – mit Hilfe von template reference variables – verwendet.
- Wir können auch eine Bindung auf eine im .ts-File definierte Variable herstellen:

```
// app.component.ts  
myVar = 'abc';
```

```
<!-- app.component.html -->  
myVar: <input [(ngModel)]="myVar">
```

Forms in Angular

Neues Event in Angular: *ngSubmit*

```
<form (ngSubmit)="logForm(firstName.value, lastName.value)">
  <input name="firstName" ngModel #firstName="ngModel" required>
  <input name="lastName" ngModel #lastName="ngModel" required>
  <button>Submit</button>
</form>
```

Forms in Angular

Neuer Forms-Controller: ngForm (analog zu ngModel für input-Elemente)

```
<form #f="ngForm" (ngSubmit)="logForm(f.value)">
  <input name="firstName" ngModel>
  <input name="lastName" ngModel>
  <button>Submit</button>
</form>
```

f.value beinhaltet ein Objekt der Form:

```
{ "firstName": "John", "lastName": "Smith" }
```

Beispiel

- app form-examples
 - component example3-form

Forms-Attribute: Überblick

<form **ngForm** ...>: fügt Controller zu einem form hinzu (eigentlich automatisch, daher nicht wirklich notwendig)

<input **ngModel** ...>: fügt Controller zu einem input hinzu

<form ngForm **#f="ngForm"** ...>: „exportiert“ den Form Controller als Template Reference Variable

<input ngModel **#firstName="ngModel"** ...>: „exportiert“ den Input Controller als Template Reference Variable

Forms: Beispiel

- Todo-App: neue Form zum hinzufügen von Todos

Forms: Beispiel

- Todo-App: neue Form mit Suchfunktion und two-way data-binding:
 - Es sollen nur Todos angezeigt werden, die den Suchtext enthalten

Services

Services allgemein

Im allgemeinen: Services sind Teile einer Angular-Anwendung, die nicht direkt mit dem „Kerngebiet“ von Angular – dem View – zu tun haben.

Beispiele:

- Datenservice
- Loggingservice
- App-Konfiguration
- Hilfsmethoden

Services haben wenig Angular-spezifischen Code.

Beispiel: Bankaccount-Service

Wir erstellen eine App zum Verwalten eines Bankkontos und wollen die zugehörigen Daten von einem Service verwalten lassen.

```
ng generate service Bankaccount
```

Dieser Befehl erstellt die Klasse *BankaccountService*.

Services und Dependency Injection

Services werden in Angular nicht direkt von einer Komponente verwendet, sondern ihr mittels Dependency Injection im Constructor zur Verfügung gestellt. Dafür haben die Services den Dekorator @Injectable().

Das erleichtert unter anderem das Schreiben von Tests.

Beispiel:

```
// app.module.ts
import {BankaccountService } from './bankaaccount.service';
[...
  providers: [BankaccountService]
[...

// Bankaaccount.service.ts
constructor(private accountService: BankaccountService) {
  ...
}
```

Services und Dependency Injection

Wir geben an, dass wir für unsere Komponente eine Instanz der Klasse *BankaccountService* benötigen.

Angular erstellt im Hintergrund eine Instanz davon und *injiziert* diese in jede Komponente, die sie benötigt.

Beispiel: BankaccountService

In unserem BankaccountService möchten wir folgende Methoden verwenden:

```
setInitialBalance(balance)
```

```
addTransaction(description, date, amount)
```

```
getCurrentBalance()
```

HTTP

HTTP

Verschiedene Möglichkeiten, um *http* im Browser nutzen zu können:

- XMLHttpRequest
- jQuery
- Fetch

Standard in Angular: *httpClient* – Modul mit „Observables“

HTTP in Angular: Observables

Observables in Angular: Möglichkeit, asynchron Daten abzufragen –
Ähnlich zu *Promises*.

HTTP in Angular: Observables

```
this.http.get('https://jsonplaceholder.typicode.com/todos')  
  // .subscribe() ... ähnlich zu .then() bei promises  
  // wir senden einen Request, wenn dieser beantwortet wird,  
  // wird die Arrow-Funktion aufgerufen und das Resultat  
  // unter .todos gespeichert  
  .subscribe((response) => {  
    this.todos = response;  
  })
```

HTTP in Angular: Einbindung

```
// app.module.ts:
```

```
import { HttpClientModule } from '@angular/common/http';
```

```
...
```

```
imports: [..., HttpClientModule]
```

```
// app.component.ts oder todo.service.ts
```

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

```
...
```

Beispiel: Todo-Service

In unserer Todo-App wollen wir die Daten vom Server unter <https://jsonplaceholder.typicode.com/todos> erhalten.

Wir erstellen eine entsprechende Abfrage, die die Daten abrufen und in unser gewünschtes Format übersetzt.

Mehr zu http

<https://angular.io/guide/http>

<https://blog.angular-university.io/angular-http/>

Routing

Routing

Nutzung von HTML5-Routes (client-seitig):

<https://mywebsite.com/items/28>

Früher / für ältere Browser:

<https://mywebsite.com/#/items/28>

Achtung: Server muss entsprechend konfiguriert sein und für /items/28 das gleiche zurückliefern wie für /

Routing - Beispiel

Wir teilen unsere Todo-App in zwei *Views* auf:

- Liste aller Todos unter /
- Formular zum hinzufügen von Todos unter /*add*

Dazu definieren wir zwei neue Komponenten: *todo-list* und *add-form*

Routing – Grundlagen

Zuweisung von Komponenten zu Routen

```
// app-routing.module.ts
import { TodoListComponent } from './todo-list/todo-list.component';
import { AddFormComponent } from './add-form/add-form.component';

const routes: Routes = [
  { path: '', component: TodoListComponent },
  { path: 'add', component: AddFormComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  ...
})
```

Routing – Grundlagen

`<router-outlet>`:

Die Anzeige der Inhalte erfolgt *unterhalb* des `<router-outlet>` - Tags

Routing – Redirects und wildcards

Beispiel:

```
{ path: 'home', redirectTo: '' },  
{ path: 'add-todo', redirectTo: 'add' },  
  
{ path: '**', redirectTo: '' },
```

Routing – Links

Der Link

```
<a href="/add">add Todo</a>
```

würde zu einem Neuladen der Seite führen.

Stattdessen verwenden wir:

(nicht nur bei Links!! auch buttons z.B.)

```
<a routerLink="/add">add Todo</a>
```

Routing – Navigation aus TypeScript

Auf den Angular-Router kann in TypeScript zugegriffen werden, um die aktuell aktive Route abzufragen / zu ändern:

```
// add-form.component.ts
import { Router } from '@angular/router';
[...]  
constructor(public router: Router) { }  
  
addTodo(todo: {description: string, done: boolean}) {  
  this.todoService.addTodo(todo);  
  this.router.navigate([' / ']);  
}
```


Routing – Routen-Parameter

Wir erstellen neue Routen der Form: */todo/\$todoid*

Dort soll jeweils ein einzelnes Todo-Item angezeigt werden

```
// app.module.ts
```

```
[...]
```

```
{ path: 'todo/:todoId', component: TodoDetailsComponent },
```

Routing – Routen-Parameter

Die aktive Route bekommen wir über die Klasse *ActivatedRoute*, die wir mit Dependency Injection initialisieren.

ActivatedRoute.params ist ein *Observable* mit Routenparametern

```
// todo-details.component.ts
[...]  
export class TodoDetailsComponent {  
  todoId: string;  
  constructor(private route: ActivatedRoute, public todoService: TodoService) {  
    route.params.subscribe( (params) => {this.todoId = params.todoId})  
  }  
}
```

Routing – mehr zum Thema

<https://angular.io/guide/router>

Material Design – Komponenten

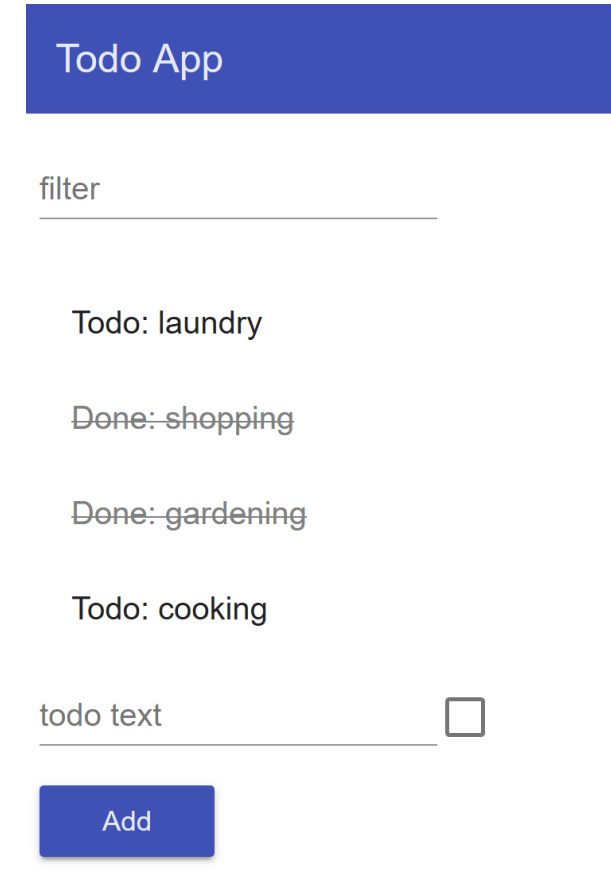
Nutzung vorgefertigter Komponenten

Material – Komponenten

- Von Google bereitgestellte Angular-Komponenten im *Material design* – Stil
- Einstieg / Setup:
 - material.angular.io – Schritte 1 - 4

Material – Übung

- Wir stellen die Todo-App auf Material Design um
- Benötigte Material-Komponenten:
 - button
 - checkbox
 - input
 - toolbar
 - list



The screenshot shows a mobile application interface for a todo list. At the top is a blue header bar with the text 'Todo App'. Below the header is a search bar labeled 'filter'. The main content area displays a list of items: 'Todo: laundry', 'Done: shopping', 'Done: gardening', and 'Todo: cooking'. At the bottom, there is an input field labeled 'todo text' followed by a checkbox and a blue 'Add' button.