|  |
| --- |
|  |
| Angular 2+ |
| Tutorial am Beispiel des Projekts: todo-App |

|  |
| --- |
| Zenß Jens-Uwe  20.10.2018 |

Inhalt

[Angular installieren und ein Projekt erstellen 2](#_Toc528574035)

[Ein erster Blick in das Projektverzeichnis 3](#_Toc528574036)

[index.html 3](#_Toc528574037)

[angular.json 4](#_Toc528574038)

[app.module.ts 4](#_Toc528574039)

[main.ts 5](#_Toc528574040)

[app.component.ts, app-component.html, app-component.css…die Root-Komponente 5](#_Toc528574041)

[Struktur eines neuen Projektes 7](#_Toc528574042)

[Kommunikation innerhalb einer Komponente 8](#_Toc528574043)

[Interpolation 8](#_Toc528574044)

[Property-Binding 8](#_Toc528574045)

[Event-Binding 9](#_Toc528574046)

[Komponentenübergreifende Datenkommunikation 9](#_Toc528574047)

[Input-Binding (komponentenübergreifendes Property-Binding) 9](#_Toc528574048)

[Outputbinding (komponentenübergreifendes Eventbinding) 11](#_Toc528574049)

[Services 12](#_Toc528574050)

[Direktiven 14](#_Toc528574051)

[Pipes 16](#_Toc528574052)

[Aufgabenstellung 17](#_Toc528574053)

[Backend 17](#_Toc528574054)

[Frontend 17](#_Toc528574055)

# Angular installieren und ein Projekt erstellen

Links: <https://cli.angular.io/>, <https://angular.io/guide/quickstart>, https://code.visualstudio.com/

Voraussetzung: bereits installiertes node.js, Installation von Visual Studio Code

Die Installation des Angular Command Line Interface erfolgt über folgendes Kommando:

PS C:\Angular> npm install –g @angular/cli

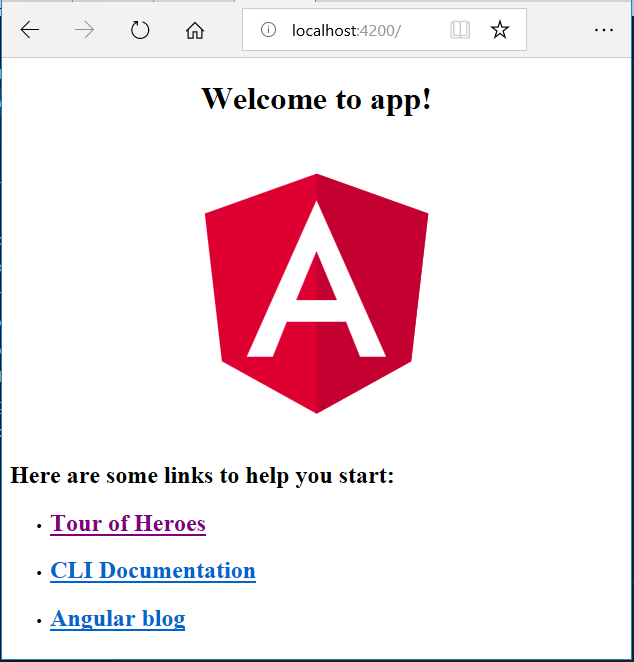
Mit ***ng new*** lassen sich neue Projekte erstellen:

PS C:\Angular> ng new todo

Zum Starten des Projektes kann mit ***cd*** in das Verzeichnes des Projektes gewechselt werden und mit ***ng serve*** das Angular-Projekt gestartet werden.

PS C:\Angular> cd todo

PS C:\Angular\todo> ng serve

Standardmäßig kann dann unter <http://localhost:4200> die Angular-Seite aufgerufen werden. Die Startseite sollte dann so aussehen.

# Ein erster Blick in das Projektverzeichnis

## index.html

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Todo</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

</html>

Diese Datei ist der Startpunkt der Applikation und befindet sich im ./src-Verzeichnis. Von hier aus wird über den ***Selektor-Tag***

<app-root>

die Startkomponente ***AppComponent*** aufgerufen. Möchte man eine andere Startkomponente auswählen, muss man hier den Selektor anpassen und in der ***app.module.ts*** muss die neue Startkomponente im Eintrag:

bootstrap: [AppComponent]

geändert werden. Es werden keine CSS-Dateien oder JavaScripts in der index.html bekanntgegeben, dies geschieht in der Datei angular.json im root-Verzeichnis.

## angular.json

Für unser erstes Projekt ist erst einmal nur wichtig: Hier wird die globale CSS-Datei festgelegt (***styles.css*** im src-Verzeichnis).

{

*"$schema"*: "./node\_modules/@angular/cli/lib/config/schema.json",

*"version"*: 1,

*"newProjectRoot"*: "projects",

*"projects"*: {

*"todo"*: {

*"root"*: "",

*"sourceRoot"*: "src",

*"projectType"*: "application",

*"prefix"*: "app",

*"schematics"*: {},

*"architect"*: {

*"build"*: {

*"builder"*: "@angular-devkit/build-angular:browser",

*"options"*: {

*"outputPath"*: "dist/todo",

*"index"*: "src/index.html",

*"main"*: "src/main.ts",

*"polyfills"*: "src/polyfills.ts",

*"tsConfig"*: "src/tsconfig.app.json",

*"assets"*: [

"src/favicon.ico",

"src/assets"

],

*"styles"*: [

"src/styles.css"

],

*"scripts"*: []

},

CSS-Dateien wie zum Beispiel bootstrap.min.css werden unter „styles“ eingetragen, auszuführende Javascripts wie z.B. jquery.js unter „scripts“.

## app.module.ts

Angular Apps sind modular und Angular hat ein eigenes Modularitätssystem namens ***NgModules***. ***NgModule*** sind dabei Container für einen zusammenhängenden Codeblock, bei denen unter anderem ***Components,*** ***Services, Directives oder Pipes*** oder Untermodule zusammengefasst und für andere Module oder Komponenten zur Verfügung gestellt werden können.

Jede Angular-App hat mindestens eine NgModule-Klasse, das ***Root-Modul***, das üblicherweise als ***AppModule*** bezeichnet wird und sich in einer Datei mit dem Namen app.module.ts befindet.

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export *class* AppModule { }

* ***declarations:*** Die Komponenten, Direktiven und Pipes, die zu diesem NgModule gehören, werden hier bekanntgegeben.
* ***exports:*** Die Teilmenge der Deklarationen, die in den Komponentenvorlagen anderer NgModule sichtbar und verwendbar sein sollen. (hier nicht verwendet)
* ***imports:*** Andere Module, deren exportierte Klassen von in diesem NgModule deklarierten Komponentenvorlagen benötigt werden. (hier nicht verwendet)
* ***providers:*** Bekanntgabe von Diensten (***services***), die dieses NgModule zur globalen Sammlung von Diensten beiträgt; Sie werden in allen Teilen der App zugänglich.
* ***bootstrap:*** Die Hauptanwendungsansicht, die Root-Komponente, die alle anderen App-Ansichten hostet. Nur das ***Root-Modul*** sollte die Bootstrap-Eigenschaft festlegen.

## main.ts

Der echte Start der Applikation erfolgt in der Datei main.ts durch die Übergabe des definierten Moduls an die gewünschte Plattform:

import { enableProdMode } from '@angular/core';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(*err* *=>* *console*.log(err));

## app.component.ts, app-component.html, app-component.css…die Root-Komponente

app.component.html:

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{ title }}!

</h1>

<img width="300" alt="Angular Logo" src="data:image/svg+xml;base64,PHN2………“>

</div>

<h2>Here are some links to help you start: </h2>

<ul>

<li>

<h2><a target="\_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</a></h2>

</li>

<li>

<h2><a target="\_blank" rel="noopener" href="https://github.com/angular/ang...">

CLI Documentation</a></h2>

</li>

<li>

<h2><a target="\_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>

</li>

</ul>

Dies ist das Template der root-Komponente und wird durch den Aufruf des Selektor-Tags <app-root> in der index.html angezeigt (Startseite im Browser siehe oben).

app.component.ts

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

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

})

export *class* AppComponent {

title = 'app';

}

Auffallend ist sofort, das die Klasse AppComponent einen ***Dekorator @Component( …. )*** hat. Dies ist eine Besonderheit in Angular, so werden den Klassen oder auch einzelnen Variablen ***Metadaten*** übergeben, hier in dem Fall der ***Selektor***, der den Tagnamen für die Verwendung in Templates bekanntgibt, das zugehörige ***Template*** app.component.html und die zugehörige ***CSS-Datei*** app.component.css. Alternativ können auch durch Verwendung von template anstatt templateURL oder styles anstatt stylesURLs inline die Daten als String angegeben werden.

Die app.component.css ist standardmäßig leer.

# Struktur eines neuen Projektes

angular.json

*…*

*"styles"*: [ "src/styles.css"

],

*"scripts"*: []

…

Zu importierende CSS- und JS-Komponenten mit globaler Gültigkeit

CSS, JS

global!

<!doctype html>

<html lang="en">

<head>

…

</head>

<body>

<app-root></app-root>

</body>

</html>

HTML

Index.html

Festlegung des ***root-Moduls***

main.ts

Festlegung der ***root-Komponente*** mit

bootstrap: [AppComponent]

app.module.ts

App

Root-Applikation

HTML-Syntax mit [poperty

<template>

app.component.html

CSS

app.component.css

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

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

})

@Component

AppComponent

# Kommunikation innerhalb einer Komponente

## Interpolation

App

<h1>

Welcome to {{ title }}!

</h1>

<template>

AppComponent

export *class* AppComponent {

title = 'app';

}

@Component

AppComponent

Bei Interpolation findet der Datenaustausch in einer Richtung vom Component zum Template statt. Im Template wird der zu ersetzende Ausdruck durch {{ }} gekennzeichnet und kann ein Wert oder auch eine Funktion sein, die einen Wert oder ein Objekt zurückliefert.

## Property-Binding

App

export *class* AppComponent {

test = true;

getTest(){return this.test}

}

@Component

<button [innerText]="getTest()"></button>

<!--oder andere Schreibweise-->

<button bind-innerText="getTest()"></button>

<template>

Der Datenaustausch erfolgt auch in eine Richtung vom Component zum Template. Im Template wird das „DOM-Property“ durch [ ] gekennzeichnet. Es muss darauf geachtet werden, dass HTML-Attribute nicht gleich DOM-Propertys sind, so z.B. ist das HTML-Attribut colspan in <td> nicht als DOM-Property bekannt. Beispiel für Attribut-Property:

<td [attr.colspan]="getNumber()"></td>

## Event-Binding

App

export *class* AppComponent {

eventClick(*event*:Event){

*let* b=event.target as HTMLButtonElement

...

}

}

@Component

<button (click)="clickEvent($event)"></button>

<!--oder andere Schreibweise-->

<button on-click="clickEvent($event)"></button>

<template>

Der Datenaustausch erfolgt in Richtung vom Template zum Component. Im Template wird der Event-Listener durch ( ) gekennzeichnet. Es wird ein Funktionsname mit Eingabeparameter aus der Component-Klasse angegeben, die das Event abarbeitet. Alternativ kann auch ein Ausdruck direkt im Template angegeben werden. Das Element, welches das Event ausgelöst hat, kann mit event.target übergeben werden werden.

# Komponentenübergreifende Datenkommunikation

## Input-Binding (komponentenübergreifendes Property-Binding)

Child

Parent

<p>

zu Child: {{content}}

</p>

<template>

<div>

<h1>Parent</h1>

<app-child [content]="toChild">

</app-child>

</div>

<template>

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

@Component(. . .)

export *class* ChildComponent implements *OnInit* {

@Input() content: *string*

ngOnInit() {}

}

@Component

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

@Component( . . .)

export *class* ParentComponent implements *OnInit* {

toChild: = "Immer schön artig sein!"

ngOnInit() {}

}

@Component

Die Datenübertragung findet beim Inputbinding vom Parent-Template zur Child-Component-Klasse statt. Die Syntax entspricht dem des normalen Propertybinding und wird am aufrufenden Selektor-Tag angewendet. Die Bezeichnung der Eigenschaft im Template und der Variable im child müssen identisch sein und diese mit dem @Input() Decorator gekennzeichnet sein. Dabei muss der Dekorator über das Schlüsselwort Import von @angular/core importiert werden.

So kann hier am Beispiel letztendlich der Inhalt der Variable toChild in der Parent-Component-Klasse auf die Variable content im Child übertragen und dort schlussendlich im Template über Interpolation angezeigt werden.

# Lösung:

Eine Lösung kann mit folgendem Befehl: git clone <https://github.com/jens-uwe/todo> heruntergeladen werden.

## Outputbinding (komponentenübergreifendes Eventbinding)

Parent

<div><h1>Parent</h1>

<p>Frage: {{ contentFrage }}</p>

<app-child [frage]="contentFrage" (antwort)="contentAntwort($event)"> </app-child>

<p> Antwort: {{antwort}}</p>

<input type="text" #parentFrage>

<button (click)="starteFrage(parentFrage.value)">Frage</button>

</div>

<template>

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

@Component(. . .)

export *class* ParentComponent implements *OnInit* {

contentFrage = ""

antwort = ""

starteFrage(*eingabe*: *string*){

this.contentFrage = eingabe

}

contentAntwort(*eingabe*: *string*){

this.antwort = eingabe

}

}

@Component

Child

<div>

<h1>Child</h1>

<p>Frage vom Elter: {{frage}}</p>

</div>

<template>

import { Component, OnInit, Input, EventEmitter, Output, OnChanges } from '@angular/core';

@Component(. . .)

export *class* ChildComponent implements *OnInit*, *OnChanges* {

@Input() frage: *string*

@Output('antwort') antwort: EventEmitter<*string*>

*constructor*() {

this.antwort = new EventEmitter()

}

ngOnChanges() {

if (this.frage.includes("höre")) this.antwort.emit("Nö")

else (this.antwort.emit("Nein"))

}

}

@Component

Hier am Beispiel wird zunächst über Inputbinding eine Frage vom Parent an das child übertragen. Ziel ist es, der Elternkomponente sofort eine Antwort zu übermitteln. Hierbei wird die Funktion ngOnChanges() verwendet, welche immer dann ausgeführt wird, wenn eine Änderung von gerenderten Objekten in der Kindskomponente stattfindet (hier ist das die neue Frage der Elternkomponente). Zum Verwenden dieser Funktion muss das zugehörige Interface von @angular/core importiert werden und mit dem Schlüsselwort implements OnChanges bekanntgegeben werden.

Das Outputbinding ist ein Eventbinding, dass heißt: Es muss ein benutzerdefiniertes Eventbinding in der Child-Component-Klasse ausgelöst/gesendet werden.

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

@Component(. . .)

export *class* ChildComponent implements *OnInit*, *OnChanges* {

..

@Output('antwort') antwort: EventEmitter<*string*>

*constructor*() {

this.antwort = new EventEmitter()

}

ngOnChanges() {

if (this.frage.includes("höre")) this.antwort.emit("Nö")

else (this.antwort.emit("Nein"))

}

Wir benötigen daher zuerst den Import von Output und EventEmitter aus @angular/core.

Als nächstes deklarieren wir das Output-Event mithilfe des @Output-Dekorators. Als Eingabe wird im Dekorator der Name des Events, so wie er im perent-Template aufgerufen werden soll, angegeben. Weiterhin müssen wir bei EventEmitter auch den Datentyp der zu versendenden Nachricht angeben.

Im Konstruktor müssen wir dann eine Instanz von unserer EventEmitter-Klasse bilden.

Nun kann bei allen auftretenden Änderungen bei @Input frage das Event mittels der Funktion emit(Nachricht) abgefeuert werden, da im Falle einer Änderung immer ngOnChanges aufgerufen wird.

Nachdem das Event im parent-Template angekommen ist, kann es ganz normal in der Elternklasse abgearbeitet werden.

## Services

Der Service ist eine weitere Kategorie in Angular, die Werte und Funktionen umfasst und diese für andere Komponenten zur Verfügung stellt. Ein Service ist typischerweise eine Klasse mit einem engen, genau definierten Zweck.

Angular unterscheidet Komponenten von Services, um die Modularität und Wiederverwendbarkeit zu erhöhen. Indem die sichtbezogene Funktionalität einer Komponente von anderen Verarbeitungsarten getrennt werden, können Komponentenklassen schlanker und effizienter gemacht werden.

Eine Komponente kann bestimmte Aufgaben an Dienste delegieren, z. B. das Abrufen von Daten vom Server, das Validieren von Benutzereingaben oder das direkte Anmelden an der Konsole. Indem Sie solche Verarbeitungsaufgaben in einer injizierbaren Serviceklasse definieren, stellen Sie diese Aufgaben für jede Komponente zur Verfügung.

Mit folgendem cli-Kommando können Service-Klassen erzeugt werden:

PS C:\Angular\service> ng generate service servicename

Es wird dann im ./src-Verzeichnis (Standard) der App die Datei servicename.service.ts angelegt

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

@Injectable({

({

providedIn: 'root'

}

})

export *class* ServiceService {

*constructor*() { }

}

Durch {providedIn: 'root'} wird der Service dem Modul bekanntgegeben. Alternativ kann der Service auch ohne {providedIn: 'root'} angelegt werden. Dann allerdings muss man sich um den Eintrag in der app.module.ts selbst kümmern, er wird unter providers angemeldet und vorher natürlich importiert.

. . .

import { ServicenameService} from './service.service'

@NgModule({

declarations: [

AppComponent,

EingabeComponent,

AusgabeComponent

],

imports: [

BrowserModule

],

providers: [

ServicenameService

],

bootstrap: [AppComponent]

})

export *class* AppModule { }

Die Nutzbarmachung des Services für eine Komponente ist denkbar einfach und erfolgt mittels Dependency Injection im Konstruktor der Komponente:

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

import {ServiceService} from '../service.service'

@Component({

selector: 'app-eingabe',

templateUrl: './eingabe.component.html',

styleUrls: ['./eingabe.component.css']

})

export *class* EingabeComponent implements *OnInit* {

eingabe=""

*constructor*(public *service*:ServiceService) { }

eingabeEvent(*eingabe*: *string*){

this.service.setPuffer(eingabe)

}

}

Nun können über die im Konstruktor festgelegt Variable auf alle (öffentlichen!) Eigenschaften und Funktionen der Serviceklasse zugegriffen werden.

## Direktiven

Direktiven sind Komponenten ohne eigenes Template. Die Aufgabe eine Direktive besteht daher in der Manipulation des Verhaltens eines Templates oder genauer gesagt, einzelner DOM-Elemente des Templates. Klassen von Direktiven werden mit dem Dekorator ***@Direktive*** gekennzeichnet. Wir haben bereits von Angular vordefinierte strukturelle Direktiven verwendet und zwar: ***\*ngIf*** und ***\*ngFor***

<tbody>

<tr \*ngFor = "let person of db.persons">

<td>{{person.pid}}</td>

<td>{{person.firstname}}</td>

<td>{{person.lastname}}</td>

</tr>

</tbody>

Hier wird je nach Anzahl der Personen im Array db.persons <tr> mit den dazugehörigen Childs dynamisch mehrfach dargestellt.

Mit ***\*ngIf*** kann ich entscheiden, ob ein HTML-Element überhaupt angezeigt wird oder nicht.

Erzeugt wird unsere Direktive mit:

PS C:\Angular\service> ng generate directive test

Es wird automatisch die Datei ***test.directive.ts*** erzeugt:

import { Directive } from '@angular/core';

@Directive({

selector: '[appTest]'

})

export *class* TestDirective {

*constructor*() { }

}

Weiterhin kümmert sich der Befehl darum, dass die Direktive in der ***app.module.ts*** unter ***declarations*** bekanntgegeben wird.

Am folgenden Beispiel wird das <td>-Tag im Verhalten um Funktionalitäten eines Buttons erweitert:

<tbody>

<tr \*ngFor = "let todo of db.todos | todofilter:person.pid">

<td>{{todo.tid}}</td>

<td>{{todo.text}}</td>

<td buttonview="{{todo.status}}" (setStatus)="setStatus(todo.tid, $event)"></td>

</tr>

</tbody>

Im obigen Template wird im letzten <td> bekanntgegeben, dass eine Direktive mit dem Selektor ***Buttonview*** das Verhalten ergänzen soll. Gleichzeitig wird ***Buttonview*** auch für Input-Binding in die Direktive genutzt, da hier der Wert todo.status übergeben wird. Ebenfalls haben wir über die Direktive mit Output-Binding das Event ***setStatus*** festgelegt.

import { Directive, ElementRef, Input, Output, EventEmitter } from '@angular/core';

import { DbService, Todo } from '../services/db.service'

@Directive({

selector: '[buttonview]'

})

export *class* ViewDirective {

@Input() buttonview:*number*

btnInnerText:*string*

@Output('setStatus') setStatus:EventEmitter<*number*>

*constructor*(private *elemRef*: ElementRef, private *db*:DbService) {

this.setStatus = new EventEmitter()

}

ngOnInit(){

*console*.log(this.buttonview)

this.btnInnerText = this.buttonview==0?"offen":"erledigt";

*let* btn = document.createElement("button") as HTMLButtonElement

btn.innerText = this.btnInnerText;

btn.addEventListener("click",(*event*)*=>*{

this.buttonview = this.buttonview==0?1:0

btn.innerText = this.buttonview==0?"offen":"erledigt";

this.setStatus.emit(this.buttonview)

*console*.log(this.buttonview)

});

(this.elemRef.nativeElement as HTMLElement).appendChild(btn)

}

Die Direktive manipuliert innerText von <td> und fügt so einen Button hinzu, über den dessen Event aus der Direktive zurück ins Template über Output-Binding gebracht wird. So kann die zum Template gehörende Component-Klasse über Datenänderungen informiert werden.

## Pipes

Pipes haben die Aufgabe, Werte des Datenmodells vor der Darstellung durch das Template zu transformieren. Erzeugt werden sie durch:

PS C:\Angular\service> ng generate pipe <pipename>

Es wird eine entsprechende Datei <pipename>.pipe.ts angelegt und wie bei der Direktive wird di Pipe auch unter declarations in der app.mudule.ts angemeldet.

Das Prinzip ist einfach und wirkungsvoll, es erklärt sich sehr gut an folgendem Beispiel:

Das Template, welches die Pipe todofilter verwendet, wir kennen es bereits aus dem vorigen Kapitel:

<tbody>

<tr \*ngFor = "let todo of db.todos | todofilter:person.pid">

<td>{{todo.tid}}</td>

<td>{{todo.text}}</td>

<td buttonview="{{todo.status}}" (setStatus)="setStatus(todo.tid, $event)"></td>

</tr>

</tbody>

Die Pipe:

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

import { Todo } from '../services/db.service'

@*Pipe*({

name: 'todofilter'

})

export *class* TodofilterPipe implements *PipeTransform* {

transform(*value*: Todo[], *person*: *number*): Todo[] {

return value.filter(*x=>*x.pid==person);

}

}

Sie verwendet das Interface Pipetransform und benötigt daher die Funktion transform. Value beinhaltet die Daten, welche vom Template übergeben werden, args? (hier zu person geändert) übergibt zusätzliche Parameter, die man nach dem Pipenamen im Template per Doppelpunkt übergeben kann. Hier wird letztendlich das vorhandene Todolist-Array gefiltert, in dem nur die Todos einer bestimmten Person angezeigt werden.

PS:

export *type* Todo = {

tid: *number*,

text: *string*,

status: *number*,

pid: *number*

}

# Aufgabenstellung

## Backend

Serverseitig soll mittels Express eine MySQL-Datenbank angesprochen werden, welche folgendermaßen definiert ist:

drop database if exists todo;

create database todo;

use todo;

create table person (

pid int not null auto\_increment,

firstname varchar(50) not null,

lastname varchar(50) not null,

primary key (pid)

);

create table todolist (

tid int not null auto\_increment,

text varchar(50) not null,

status *boolean* not null,

pid int not null,

primary key (tid),

foreign key (pid) references person(pid)

);

insert into person (firstname, lastname) values

("Antonia","Müller"),

("Sarah","Bäcker"),

("Anna-Maria","von Entenhausen");

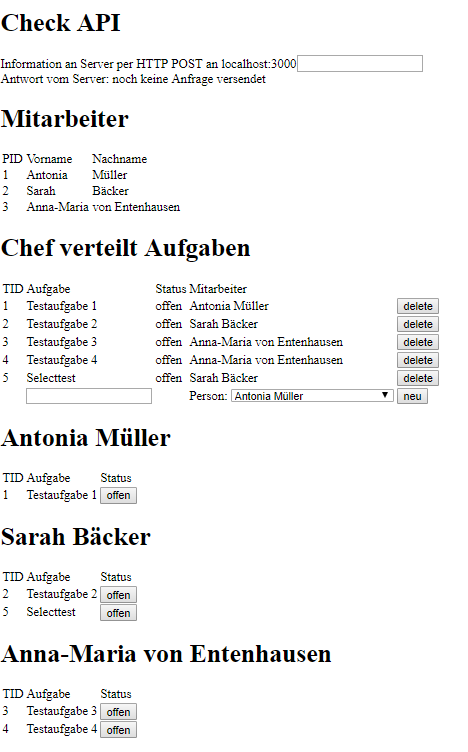
Dabei sind der Einfachheit halber die Personen vorgegeben, die Todos sollen aber durch die Frontendanwendung vom Chef erstellt und gelöscht werden können und für den Chef angezeigt werden können. Weiterhin sollen für jeden Mitarbeiter seine Todos separat angezeigt werden können und der Status gewechselt werden können.

Der Datenaustausch zwischen Frontend und Backend soll durch REST-API erfolgen (mittels http-POST-Anfragen, bzw. Antworten auf diese Anfragen)

## Frontend

Die Angular-Anwendung soll einen Service enthalten, der sich um die Kommunikation mit dem Server (REST-API) kümmert und alle Modelldaten zentral abspeichert und für mögliche Komponenten, Direktiven und/oder Pipes zur Verfügung stellt.

Die dargestellte Seite könnte folgendermaßen aussehen (in unschick ☺):



Dabei ist das Template der „Parent“-Komponente dahingehend vorgegeben, dass wie folgt Child-Komponenten genutzt werden sollen:

<h1>Check API</h1>

<label>Information an Server per HTTP POST an localhost:3000</label>

<input type="text" #testMessage>

<br>

<label>Antwort vom Server:</label>

{{http\_antwort$ | async}}

<br>

<h1>Mitarbeiter</h1>

<table>

<thead>

<tr>

<td>PID</td>

<td>Vorname</td>

<td>Nachname</td>

</tr>

</thead>

<tbody>

<tr \*ngFor = "let person of db.persons">

<td>{{person.pid}}</td>

<td>{{person.firstname}}</td>

<td>{{person.lastname}}</td>

</tr>

</tbody>

</table>

<app-todolist></app-todolist>

<div \*ngFor = "let person of db.persons">

<app-person [person]="person"></app-person>

</div>

Bei Check-API soll nur die Kommunikation mit dem Server getestet werden, im Erfolgsfall wird die Anfrage vom Server einfach zurückgesendet.