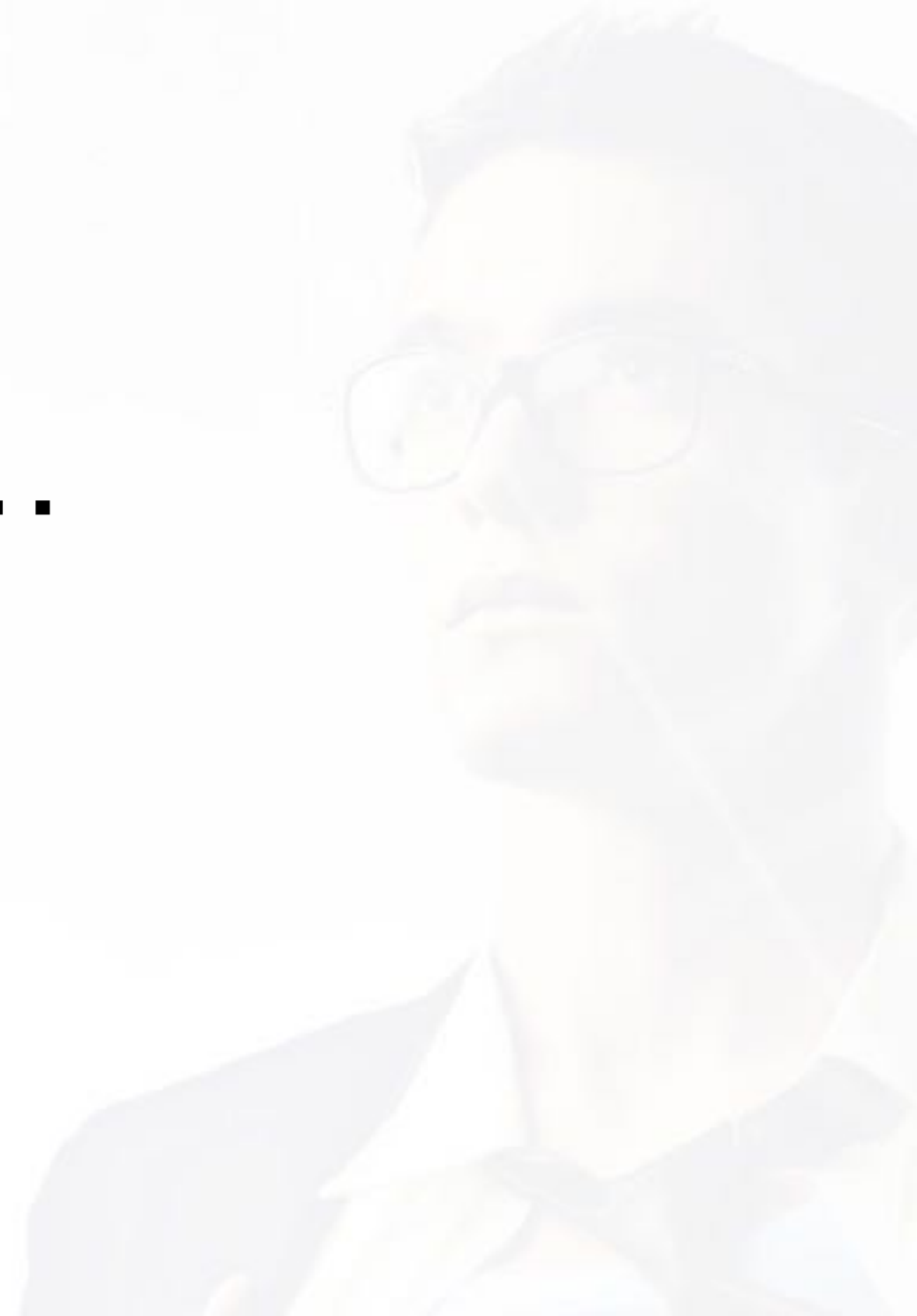


Angular Grundlagen

von: Saban Ünlü
für: HBB Technologies AG

Zwei Worte zu mir ...



Saban Ünlü

- Software Architekt und Programmierer
- Berater und Dozent seit 2000
- Autor
- Influencer
- Gründer von netTrek



Einleitung



Einleitung

- Was ist Angular?
- Angular: Die Highlights
- Was ist neu?
- Unterschiede zu AngularJS 1x
- Technologien
- Polyfills und Vendors
- Angular-Module

Was ist Angular?

- Framework für Single Page Application
- Komponentenbasierte Anwendungen
– inspiriert von Web-Components
- Modulare Arbeitsweise
- Trennung von Logik und View

Angular: Die Highlights

- Vorlagen
- Bindungen
- Services
- Dependency Injection
- Routing
- Formulare



Was ist neu?

- Angular ist kein klassisches Update
- Komplett neu geschrieben
- Fokus
 - Performance (3-5 mal schneller als Angular 1)
 - Komponenten
 - Modulare Arbeitsweise

Unterschiede zu AngularJS 1x

- Komponenten statt Kontroller
Wie Element-Direktiven in Angular 1
- Verzicht auf Scope
Bindung erfolgt auf Komponenten-Instanz

Unterschiede zu AngularJS 1x

- Entwicklung mit ES5, Dart oder TypeScript
- Angular wurde mit TypeScript entwickelt
 - ES2015: Klassen, Interfaces und Vererbung
 - ES2015: Templates
 - Typisiert
 - Annotations

Unterschiede zu AngularJS 1x

- Angular ist mit TypeScript ES2015 konform
- TypeScript wird für aktuelle Browser in ES5 ausgegeben
- ES2015 Polyfills für “weniger moderne” Browser
- System zum Laden und Verwalten von Modulen z.B.:
 - System.js
 - webpack

Technologien - Überblick



core js

SystemJS

RxJS

reflect-metadata

Technologie



- JavaScript-Laufzeitumgebung für diverse Betriebssysteme
- Versionierungssystem für Software (GitHub – Filehoster)
- Auf ES2015 basierende Programmiersprache

Polyfills

- core-js
 - ES2015/ES6 Polyfills
- web-animations-js
 - Firefox Animationen
- intl
 - I18n für Internationalisierung

Vendors

- RxJS
 - Bibliothek, um Ereignisse und asynchrone Prozesse zu überwachen. Wird für HTTP-Aufrufe von Angular genutzt.
- zone.js
 - Ähnlich Domains in Node: Ausführungskontext ermöglicht, die Ausführung zu überwachen und zu steuern.

Vendors

- reflect-metadata
 - Metadaten in konsistenter Weise zu einer Klasse hinzufügen
- systemjs
 - Modul Loader für ES2015/ES6 Module

Angular Module

- @angular/core
 - Notwendig für jede Anwendung – Kern für Komponenten, Direktiven, Dependency Injection und Komponentenlebenszyklus
- @angular/common
 - Häufig verwendete Direktiven, Pipes und Services

Angular Module

- @angular/compiler
 - Kombiniert Logik mit Vorlagen – Compiler wird automatisch über platform-browser-dynamic angestoßen
- @angular/platform-browser
 - Browser und DOM-relevante Bestandteile, vor allem zum Rendern neuer Elemente

Angular Module

- `@angular/platform-browser-dynamic`
 - Verfügt über die Bootstrapping-Methode
- `@angular/http`
 - Modul für HTTP-Aufrufe
- `@angular/router`
 - Module für den Komponenten-Router

Angular Module

- @angular/animate
 - Animationen im Angular-Kontext

Projektsetup



Erste Schritte

- Mac
 - XCODE installieren
 - node.js installieren ($\geq 6.9.x$)
- Win
 - node.js installieren ($\geq 6.9.x$)
 - Git installieren (inkl. Bash)

Setup Manuell

- Node initialisieren
- Abhängigkeiten installieren
- TypeScript konfigurieren
- SystemJS konfigurieren

Seed Setup

- `git clone https://github.com/angular/quickstart.git myProject`
- `npm install`

angular-cli

- `ng new hb --prefix=hb --style=scss --routing=true`
- `ng serve`
- `ng g m commonUi`
- `cd common-ui/`
- `ng g c user`

Architektur



Architektur

- Module
- Komponenten
- Bootstrap
- Direktiven
- Pipes
- Datenbindung
- Dependency Injection (DI)
- Services
- Router

Module

- Nicht vergleichbar mit JavaScript-Modulen
- Funktionen und Features in einer Black-Box bündeln
- Anwendung und eigene Module mit externen Modulen erweitern
- Compiler mitteilen, nach welchen Elementen auszuschauen ist

Module

Angular-eigene Module

- BrowserModule (Ereignisse, DOM)
- CommonModule (Direktiven, Pipes)
- HttpClientModule (XHR)
- FormsModule (Formulare)
- RouterModule (Komponenten-Router)

Module

Module erzeugen

- Modul-Klasse anlegen



Module



```
class AppModule {}
```

Module

```
@NgModule({  
  imports:    [ BrowserModule ]  
})  
  
export class AppModule {}
```


Module

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

Angular Module

- ng g m commonUi in src/app
- @NgModule
 - imports
 - definiert Module die in diesem Modul benötigt werden
 - declarations
 - benötigte Komponenten, Direktiven, Pipes

Angular Module

- @NgModule
 - providers
 - Bestimmt welche Service der Injector dieses Moduls für die DI bereitstellt.
 - exports
 - Exportiert Komponenten, Direktiven, Pipes dieses Moduls damit importierende Module das nutzen

Angular Module

- @NgModule
 - bootstrap
 - Komponenten, die beim Bootstrap dieses Moduls in den ComponentFactoryResolver abgelegt werden.
Analog - entryComponents

Angular Module

- @NgModule
 - entryComponents
 - Kompiliert Komponenten bei der Definition des Moduls. Anschließend ist die Nutzung ohne Komponente-Kontext möglich, weil es als ComponentFactory und die componentFactoryResolver abgelegt wird.

Komponenten



Komponenten

- Decorator und Metadaten
- Angular Module
- Bootstrap Root-Component
- Bootstrap eine Modules
- Selector
- Vorlagen
- Styling
- Komponenten verschachteln (Shared-Modules)
- ng-content
- ViewChilds
- Lifecycle hook

Komponenten

- Eigene HTML Knoten
- Bestandteile
 - Vorlage
 - Style
 - Logik

Komponenten

```
class AppComponent {  
  
    constructor () {  
        console.log ( "App Component" );  
    }  
  
}
```

Komponenten

```
<h1 (click)="onClick()">{{name}}</h1>
```

Komponenten

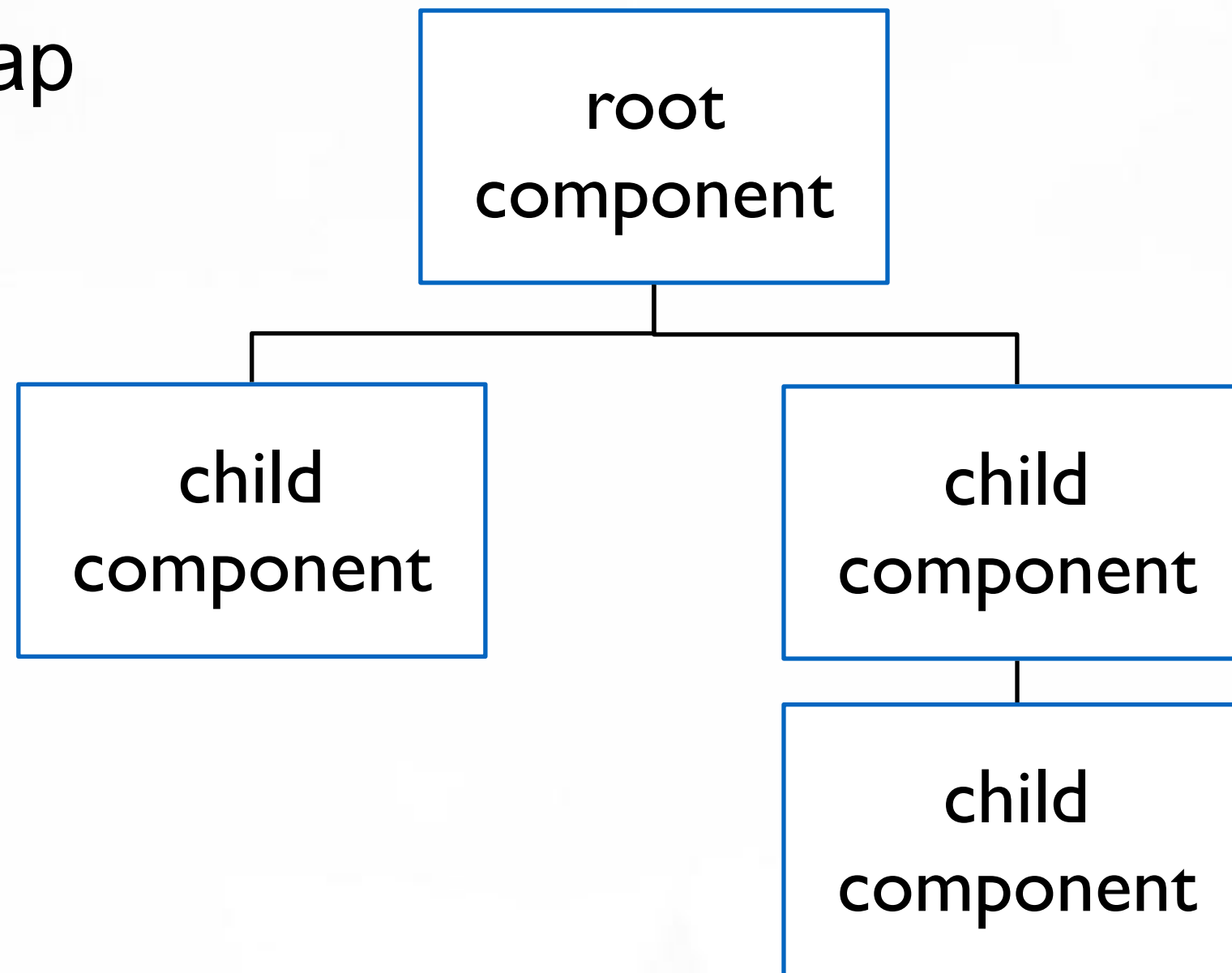
```
<h1 (click)="onClick()">{{name}}</h1>  
<my-component> </my-component>
```

Decorator und Metadaten

- Dekoratoren bereichern eine Klasse mit Metadaten
- Vor Klassendefinition
- Funktionsaufruf beginnend mit einem @NAME
- Parameter
 - Metadaten

Komponentenbasierte Anwendung

- bootstrap



Bootstrap

- in der main.ts
- platformBrowserDynamic
 - bootstrapModule
 - AppModule
 - bootstrap der Komponenten

Bootstrap

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

Komponent Metadaten

- ng g c user
 - selector
 - Knoten
 - Vorlage
 - templateUrl (file)
 - template (backticks)

Komponent Metadaten

- Style
 - styleUrls (filelist)
 - styles (backtick-list)
- Spezieller Style
 - :host
 - >>>

Komponent Metadaten

- Style
 - encapsulation - Umgang mit Webkomponenten
 - ViewEncapsulation.Emulated
 - ViewEncapsulation.Native
 - ViewEncapsulation.None

Komponent Metadaten

- Inhalte Transklusieren (transclude)
 - ng-content
 - Knoten in Vorlage
 - Attribut
 - `select="nt-table-caption"`

Komponent Content

- @ContentChild
 - Kind-Komponente einer Komponente erreichen
 - Parameter
 - Komponentenklasse
- ngAfterContentInit
 - Hook ab wann der Wert erreichbar ist.

Komponent Content

- @ContentChildren
 - Analog zu ContentChild -> QueryList<Komponenten>

Komponent View

- @ViewChild
 - Knoten einer Komponente erreichen, die in der Vorlage definiert wurden.
 - Parameter
 - Komponentenkasse
 - Referenz (string) zu einer mit #NAME ausgezeichneten Knoten

Komponent View

- @ViewChild
 - ngAfterViewInit
 - Hook ab wann der Wert erreichbar ist.

Komponent Content

- @ViewChildren
 - Analog zu ViewChild -> QueryList<Komponenten>

Bindung



Bindung

- Ausdrücke interpolieren
- Eigenschaften binden
- Style-Eigenschaften binden
- CSS-Klassen binden
- Attribute binden
- Ereignisse binden
- Komponenten-Eigenschaften
- Komponenten-Ereignisse
- HostBinding

Ausdrücke interpolieren

- Ausdruck in geschweiften Klammern
 - {{ AUSDRUCK }}
- Erlaubte Ausdrücke
 - Eigenschaften, Zeichenketten, Operatoren
 - Methodenrückgabe

Eigenschaften

- Erlaubt Zuweisung über Eigenschaften eines HTML-Elementes
- [EIGENSCHAFT]=„AUSDRUCK“
- Erlaubte Ausdrücke
 - Eigenschaften, Zeichenketten, Operatoren
 - Methodenrückgabe

Attribute

- Erlaubt Zuweisung über Knoten-Attribute eines HTML-Elementes
- [attr.EIGENSCHAFT]=„AUSDRUCK“
- Erlaubte Ausdrücke
 - Eigenschaften, Zeichenketten, Operatoren
 - Methodenrückgabe

Styles

- Erlaubt Zuweisung über StyleEigenschaften eines HTML-Elementes
- [style.EIGENSCHAFT.EINHEIT]=„AUSDRUCK“
- Erlaubte Ausdrücke
 - Eigenschaften, Zeichenketten, Operatoren
 - Methodenrückgabe

Class

- Erlaubt styling über CSS-Klassen
 - `[class.KLASSENNAME]=„BOOL-AUSDRUCK“`
 - `[class]=„AUSDRUCK“`
- Erlaubte Ausdrücke
 - Eigenschaften, Zeichenketten, Operatoren
 - Methodenrückgabe

Ereignis

- Erlaubt Bindung an native Ereignisse eines HTML-Elementes
 - (EVENT)=„METHODE(\$PARAM)“
- Parameter
 - \$event -> reicht Ereignis durch
- Beispiel
 - (click)=„clickHandler(\$event)

Eigenschaft

- Komponenteneigenschaften lassen sich über den Eigenschaftsdekorator anlegen
 - `@Input (OPT_ATTR_NAME) NAME : Type`
- Auch für Setter nutzbar
- Beispiel
 - `@Input('selected-usr') selectedInd: number = 1;`
 - `<comp [selected-usr]=„2“`

Ereignisse

- Komponentenergebnisse lassen sich über den Eigenschaftsdekorator anlegen
 - `@Output (OPT_ATTR_NAME) NAME : EventEmitter`
- Auch für Getter nutzbar
- EventEmitter sendet Wert via emit
- Wenn dem NAME der Ausdruck Change folgt ist eine Bidirektionale Bindung möglich.

HostBindings- und Listener

- Mittels Eigenschaftsdekorator lassen sich auch Bindungen direkt in der Komponentenkasse definieren
 - `@HostBinding (bind) NAME : boolean = true`
 - `@HostListener (EVT_NAME, [,$event']) HANDLER : Function = (evt)=>{ }`

Direktiven



Direktiven

- Definition
- Hauseigenen
 - ngIf
 - ngFor
 - ngClass und ngStyle
- Eigene Direktiven

Direktiven

- Direktiven sind „Komponenten ohne Vorlagen“ und werden als Attribut verwendet
- Typen
 - Strukturell
 - Modifiziert DOM
 - Attribute
 - Modifiziert die Funktionalität

Direktiven

- Selector bestimmt, wie Direktiven angewandt werden
 - Attribut `<div selector ...`
 - Mit Bindung `<div [selector]="wert"`
 - Mit Zuweisung `<div selector="wert"`
 - Klasse `<div class="selector" ...`

Strukturelle Direktiven

- modifizieren den DOM
- sind mit Asterix * oder über einen Template-Knoten nutzbar
 - [ngIf]=„AUSDRUCK“
 - Hängt den Knoten aus dem DOM wenn der Ausdruck false ist

Strukturelle Direktiven

- [ngFor]=„AUSDRUCK“
 - Wiederholt den Knoten anhand einer Iteration
 - Ausdruck
 - Beschreibt Iterator und kann zusätzliche Werte durchreichen
 - index, first, last, even, odd

Attribute Direktiven

- [ngClass]=„AUSDRUCK“
- [ngStyle]=„AUSDRUCK“
 - Erweitert style und class Attribut eines Knotens

Direktive erstellen

- @Directive
 - selector
 - Attribut z.B. [,myDirective']
 - Klasse z.B. „my-class‘ (auch als Liste)
 - class optional mit DI von ElementRef
 - nativeElement - Referenziert dann das Element

Pipes



Pipes

- Definition
- Hauseigene Pipes
- Pipes benutzen
- Eigene Pipes erstellen

Pipes

- Modifiziert die Ausgabe
- Syntax
 - Ausdruck | PipeName : Parameter
- Beispiel
 - {{name | uppercase}}

Pipes

- Hauseigene
 - Uppercase
 - Lowercase
 - Date
 - ...

Pipes

- Verwenden
 - Ausdruck | PipeName : Parameter
- Beispiel: {{name | uppercase}}
- Im Code
 - ```
const reversePipe : ReversePipe = new
ReversePipe ();
console.info (reversePipe.transform(123));
```



# Pipes

- Erstellen
  - @Pipe
    - name: string
- class NAME implements PipeTransform
  - transform(value: any, args?: any): any {

# rxjs - vergleichbar Java-Streams

# Observer / Observable

- Observable
  - Iterierbares Objekt, welches filter- und registrierbar ist, um async. Prozesse zu verfolgen
- Observer
  - Objekt das neue Werte pushen
    - In Form eines Subject Quelle des Observable
      - `asObservable();`

# Subscription

- an Observable
  - next
  - error
  - complete
- unsubscribe
- siehe: <http://rxmarbles.com/>

# Beispiel

- `observable = Observable.range ( 1, 5 );`
- `observable.subscribe (`
  - `next => console.info ( 'next %s', next ),`
  - `error => console.info ( 'error %s', error ),`
  - `() => console.info ( 'complete' )`
- `)`

# Service



# Service

- Definition
  - DI
  - Hauseigene Service
  - Einbinden
  - Erstellen
  - Injectable
- HTTP-Service

# Service

- View-unabhängige Business-Logik
- Keine Angular-Spezifikationen
  - Außer bei Service-Konstruktoren
- Bereitstellung über DI



# Erstellen

- Klasse erzeugen
- Injectable verwenden falls Konstruktor DI benötigt
- In Provider einbinden

# Dependency Injection (DI)

```
app.component.ts:19 -> http Injected Http {}
```

# Dependency Injection (DI)

```
@NgModule({
 imports: [BrowserModule, HttpClientModule],
 providers: [MyService]
})

export class AppModule {}
```

# HTTP-Service

- HTTP-Modul importieren
- HTTP-Service injizieren
- `$http.request ( url | requestData ) ->`
  - `subscribe`
    - `next`
    - `error`

# Shortcuts

- `get(url: string, options?: RequestOptionsArgs): Observable<Response>;`
- `post(url: string, body: any, options?: RequestOptionsArgs): Observable<Response>;`
- ...

# Request

- Options
  - method : RequestMethod
  - url : string
  - headers: Headers
  - body: any
  - search: URLSearchParams

# Routing



# Router

- Basis einer Single Page Application
- Routen bestimmen, welche Komponenten angezeigt werden.
- Router-Modul wird von Angular bereitgestellt



# Routing

- Routing Vorbereitung
- Routing Konfigurieren
- Router Module
- Navigation über Router Direktiven
- Navigation über Router Service
- Childs
- Events
- CanActive
- Resolve
- Parameter
- Lazy Modules

# Erste Schritte

- Modul über RouterModule.forRoot einbinden
  - Routes
    - path
    - component
  - { useHash: false }
- `<router-outlet></router-outlet>` einbinden

# redirect

- initial
  - path: "",  
pathMatch: 'full',  
redirectTo: 'list'
- 404
  - path: '\*\*',  
redirectTo: 'list'

# routerLink

- Direktive
  - Wert
    - Pfad | [ Pfad, ...Params]
- routerLinkActive
  - Wert
    - CSS-Klassenname

# Navigation über RouterService

- Router Service Injizieren
- router.navigate Methode
  - Params
    - Liste
      - Pfad
      - Parameter

# Parameter

- Route mit Parameter definieren
  - path: 'details/:id',  
component: UserDetailsComponent
- In Komponente ActivatedRoute Service injizieren
  - `this.subscription = this.route.paramMap  
 .map ( paramMap => paramMap.get ('id') || 'None' )  
 .subscribe( id => this.param_id = id );`

# Resolve

- Daten vor Routenwechsel beschaffen
- ResolveService auf Basis des Resolve Interface anlegen, einbinden und in Route einbinden
  - `path: 'details/:id',`  
`component: UserDetailsComponent,`  
`resolve: {`  
    `user: ResolveService`  
`}`

# Resolve

- Observable des ResolveService befüllt die Eigenschaft
- Abruf der Daten auf ngInit über ActivatedRoute Service
- `this.route`
  - `.data`
  - `.map ( data => data['user'] )`
  - `.subscribe( user => this.user = user );`



# CanActive

- Genehmigung der Aktivierung einer neuen Route
- Hierfür wird ein auf dem CanActive-Interface basierender Service erstellt und eingebunden
  - `canActivate ( route : ActivatedRouteSnapshot, state : RouterStateSnapshot ) : Observable<boolean>|Promise<boolean>|boolean`

# CanActive

- Service wird in die Routendefinition implementiert
  - path: 'home',  
component: HomeComponent,  
canActivate: [ CanActiveService ]

# Ereignisse

- Router Service injizieren
- events Observable<Event> subscriben
  - constructor ( router: Router ) {  
    router.events.subscribe( event => console.log  
    (event));  
}

# Child

- Eine Route kann Unterrouuten haben
- Diese müssen in der Config unter der Eigenschaft
  - children
    - analog zur vorhanden Konfiguration angelegt werden.

# Lazy Module

- loadChildren ermöglich im CLI Kontext die einfache Umsetzung
- path : 'dash',  
loadChildren: './dash/dash.module#DashModule'
  - Der Pfad zu dem Modul und der Klassenname muss übergeben werden
    - PFAD#MODUL\_NAME

# Lazy Module

- Im Modul selbst wird die Route mit der darzustellenden Komponente definiert
- RouterModule.forChild ( [  
    {  
        path : "",  
        component: DashComponent  
    }  
])

# FORM



# Formulartypen

- Template Getrieben
  - FormControls werden mittels Direktiven Erstellt
  - Daher Asynchrone
- Reactive-Forms
  - FormControls werden im Code erstellt und mittels Direktiven im Template referenziert
  - Daher Synchrone



# Bootstrap

- `#f="ngForm"`
  - Formular initialisieren und Controller referenzieren
    - `form: FormGroup`
      - `Controls: FormControl`
- FormGroup erstellen
  - Zuweisen über FormGroup Direktive

# Direktive

- `ngModel #username=„ngModel“`
  - Model Zuweisen - asny. wird der Controller erstellt und ist privater Bestandteil der NgModel-Instanz
- `ngModel`
  - Direktive für Formularelement, damit Eingaben direkt ins Model geschrieben werden
  - Und Werte aus dem Model dargestellt werden

# Direktive

- `formControlName="username"`
  - Reactive wird lediglich die FormControlName Direktive verwendet und verweist auf den Key des zuvor erstellten Controllers

# Direktive

- Gruppen erstellen
  - Template
    - ngModelGroup Direktive
      - ngModelGroup="account"
  - Reactive
    - Key mit formGroupName Direktive zuweisen

# Submit

- (ngSubmit) Ereignis auf Form registrieren
- button:submit dem Formular hinzufügen

# Klassensbasierte Validierung

- Angular hängt validierungsabh. CSS-Klassen an ein Formularelement
  - unberührt / fokus
    - ng-untouched ng-pristine ng-invalid
  - tippen / fehler
    - ng-untouched ng-dirty ng-invalid

# Klassenbasierte Validierung

- fehlerhaft
  - ng-invalid ng-dirty ng-touched
- valide
  - ng-dirty ng-touched ng-valid

# Vorlagengetriebene Formulare

- verwenden ngForm und ngModel zur Element Definition
  - `<form #f="ngForm">`
  - `<input type="text" name="username" ngModel #username="ngModel" id="username" required minlength="4">`



# FormControl ansprechen

- `<div *ngIf="( ( username.dirty || username.touched) && username.errors )" >`
  - `<p>`
    - `{{username.dirty | json}}`
    - `{{username.errors | json}}`
    - `{{username.valid | json}}`
    - `{{f.valid | json}}`
  - `</p>`

# FormGroup anlegen

- `<div ngModelGroup="account">`
- `<input type="email" name="mail" id="mail" ngModel >`
- `</div>`