14.12.2023 - 15.12.20.23

# Angular Schulung

## Typescript

Typescript wird als javascript kompiliert und ist **typed Javascript**. Dh man „kann“ **typed variablen** verwenden. Man kann aber auch plain javascript in typescript verwenden.
Was bringt das? Beim entwickeln wird schon geprüft ob der Code syntaktisch korrekt ist. Führt zu frühzeitiger Fehlererkennung. (Was oft sehr mühsam bei javascript ist).
Union Types -> string | number
Interfaces/Klassen zur Model-Deklarierung

## Modules

Modules **definieren Codeblöcke** und **strukturieren diese zu „features“.**
Es gibt bei diesem Ansatz ein **RootModule**. Dieses beinhaltet das **bootstrap** setting. Heißt es wird **beim start ausgeführt.**
Dass heißt zb. kann man ein Module Login haben das wiederum andere Feature wie ID-Austria Login Module beinhaltet. Es beinhaltet auch Services, Directives, Pipes und Services.
Modules können auch andere Modules beinhalten aber auch exported werden. Heißt sie stellen dann Ihre Components, Services, Pipes, Directives anderen Modules zur Verfügung.
Modules können auch Lazy Loaded via Routing geladen werden -> verringert Ladezeit
Mit neueren Angular Versionen wird mehr und mehr das „Component-Standalone-Prinzip“ verfolgt

**ng generate module <name>**

## Directives

Directives sind **Klassen** die **zusätzliche Funktionalität** zu Elementen hinzufügen.

- **Components**
- **Attribute directives**
  - V**erändern die Darstellung oder das Verhalten eines Elements**
  - ngClass: hinzufügen oder entfernen von css Klassen
  - ngStyle: hinzufügen oder entfernen von styles
  - ngModel: hinzufügen von two-way-binding zu einem html element
  - ngFor
  - ngSwitch
- **Structural directives**
  - **Verändern das DOM Element durch hinzufügen oder löschen von Elementen**
  - ngIf: enabled oder disabled DOM Element
  - ngFor: For-Schleife, die DOM Elemente erstellt
  - ngSwitch: Switch - welche Bedingung zutrifft rendered ein DOM Element
- **Custom directives**
  - Werden auf einem Element gesetzt und verändern so dessen Darstellung. Man arbeitet mit Inputs oder Listeners. Man übergibt einen Parameter oder zB reagiert auf einen mouseClick oder ein mouseOver, mouseLeave Event und ändert die Farbe des Elements

## Components

Component ist im Grunde eine **View mit html-File, dem component file und einem style-File**(css/scss).
Ist definiert durch den **@Component Decorator.** Der Metadaten beinhaltet:

- **selector**: name der Komponente beim Wiederverwenden
- **template** od. **templateUrl**: inline html oder template Url
- **styles** od. **styleUrl**: inline styles od. style-File Url
- **standalone**: [true/false]
- …

14.12.2023 - 15.12.20.23

# Standalone:

**Importieren direkt andere Components**, Directives und Pipes. Sie „brauchen“ **kein Module** mehr, können aber auch in Modules inkludiert werden.

**ng generate component <component_name> —standalone**

# Services:

Services werden verwendet um **Logik aus den Components auszulagern** und Rest Calls zu bündeln. Sie helfen State Management sinnvoll einzusetzen und Objekte zu **speichern.**
**Services werden auf Component, Module oder Applikation-Ebene eingesetzt.**

**ng generate service services/data**

# Pipes

Sind spezielle Funktionen, die dabei helfen **Daten zu manipulieren und oder transformieren.** Zur Anzeige von einem Datum, manipulieren von Strings, aber auch sehr hilfreich beim Debuggen von code.
Es gibt
- Built-in-pipes
  - DatePipe, UperCasePipe, LowerCasePipe, CurrencyPipe, DecimalPipe, PercentPipe, AsyncPipe und JsonPipe, …
- Custom-pipes
  - Individuelle Pipes für spezielle Anforderungen

# Dependency Injection

Es geht um die **Bereitstellung von Components, Service etc.** für andere Teile der Applikation.
Bei Angular unterscheidet man hier zwischen DI auf **Component-Level oder auch Root Level.**
- Component Level
  - Decorater @Injectable()
- Root Level
  - Decorator @ Injectable({providedInRoot})

DI auf **Komponenten** Ebene funktioniert im Decorator mittels **providers: [..service,pipe]**
Dann ist der Service auf Component Ebene eingebunden und wird auch recycled sobald der component recycled wird.
Bei **Module Driven Applikationen** kann der Service im Module definiert werden. Ebenso via providers: [ …name] und ist dann im ganzen Module verfügbar.
Durch definieren auf **Root-Level** ist der Service für die **ganze Applikation** verfügbar.

# Routing

**Navigieren zwischen Seiten oder ersetzen von Components** in einer Angular Applikation.
Routes müssen injected werden. Das heißt man injected **routes in einem Module** oder bei einer Standalone-Based Applikation in der **AppConfig**. Routes werden definiert durch das **Type-Array Routes**. Routing verfolgt das **first-match wins Prinzip** deswegen ist die Order wichtig im routing file.

# State Management

State Management ist ein Big Point in einer Angular Applikation. Man kann den State entweder auf **Komponenten Ebene** speichern oder auf **Applikationsebene**. Dh der State wird zb beim recyceln des Components auch recyclet **oder eben ist persistent**. Beim State Management kann man sich mit **Observables und Services** helfen um die gespeicherten Daten in verschiedene Komponenten zu verteilen.

# RxJS (Reactive Extensions for Javascript)

Ist eine Library die **asynchrone Operationen** in die Applikation einbindet Mithilfe von **Observables** ladet man Daten und kann diese dann **manipulieren, mapen, filtern oder composen**.

# Observables

Wird verwendet für **Event Handling, Async Operations** und **übergeben von gespeicherten variable**n an einen oder mehrere Consumer. **Consumer werden automatisch bei einer Änderung des Observables benachrichtigt.** Was heißt das? Zb ich lade Daten diese werden dann in jedem Component das das DataService verwendet injected und eben verarbeitet je nach Component. Wichtig ist auch das Observable wieder **unzusbcriben** um dataleaks zu verhinden.
Naming convention von obseravbles ist immer das **Adden eines $ am Ende** des Variablennamens.
**WICHTIG: Observables funktionieren erst wenn diese subscribed werden.**

# Subscription

Ist ein **disposable Objekt** das uns hilf ein **Obeservable zu Releasen vom State.**

# Subjects

Ist eine Möglichkeit ein **Observable zu steuern und zu erstellen**. Via **.next(value)** kann man Daten in einem Subject setzen. Subject ist ein „**multicast“**. Dh. Alle Daten die via next gepushed werden, **bekommen das selbe Objekt übermittelt**. Deswegen eignet sich ein Subject auch perfekt für ein MessageService.

Ein anderer Type eines Observables ist das **BehaviourSubject**. BehaviourSubject kann genau **1** value halten, benötigt ein initial value und **broadcastet sofort,** sobald es transmitted wird.

Observables sind **„unicast“.** Das heißt jeder Subscriber seine eigen Observable Asuführung bekommt. Dh zb bei einem Rest call wird jedesmal **ein frischer Call** gemacht wenn man subscribe ausführt.Zum Beispiel subject ist ein **„multicast“.** Alle Subscriber bekommen das selbe und neuste Objekt.
Um Observables, welche **unsicast sind, zu einem multicast umzuwandeln** kann man **shareReplay** verwenden.

```
(shareReplay({ bufferSize: 1, refCount: false })
```

- bufferSize 1 heißt es wird immer das latest verwenden ( Buffercount )
- refCount: steuert die execution -> „false“ heißt wenn wer unsubscribed und resubscribed dann wird das aktuellste Object übermittelt -> bei true wird eine neue frische execution gemacht.

# Signals

Ist eine **Speichermöglichkeit**, auf diese sich Consumers hängen können. Signals können von **einfachen variablen bis komplexe Objekte** halten und sind **writable oder read-only**.

- **Writable signal**
  - Haben den type WriteableSignal

    **var value =signal(<value>)**
    **value.set(<value>)**
  - update(value => value +3)
- **Computed**
  - Sind **readonly signals** welche die variable von writable signals bekommen.

    Var count:writeablesignal<number> ) = signal(0)
    Count readable:signal<number> = computed(() => count() +2)

  - Computed signals werden **lazy evaluated und gespeichert.** Heißt es wird erst executed beim Aufruf und wird dann gecached. Beim erneuten Aufruf wird der cache geladen und eine Berechnung, **gecheckt ob sich das writeable verändert hat,** und dann geladen. Wenn aber count geändert wird dann versteht angular das die Kalkulation nicht mehr valid ist und beim nächsten call des computed signals wird dieses upgedated. Display state is sehr wichtig.Ist es nicht visible, dann wird ein computed signal nicht executed, erst wieder wenn sich der visibility state ändert.

# Effects

Werden bei jedem **signal change ausgeführt und upgedated.** Effects sind **ähnlich zu computed signals**, aber behalten die dependencies dynamically und **entkoppeln nicht** wie das computed signal wenn es hidden wird. Effects laufen **asynchron** und **benötigen einen „injection context“**. Unter Injection context versteht man Code, der ausgeführt wird. Dh zb. ein Constructor in einem Service oder einem Component. Alternativ kann man ein Effect auch einer variable zuweisen oder auch via eines injectors den man in den options übergibt. **Injector ist ein Interface das eine neue Instanz eines objects erzeugt.**

Wann nimmt man effects?

- Zb **Logging**: wenn man bestimmt values Monitoren will oder auch debugging von code
- **localStorage updaten**
- Andere diverse render Prozesse die man mit der normalen Syntax nicht abbilden kann
-

Wann nicht?

- Man sollte **state changes** zb. Updaten **von Signals** oder anderen State Management Zyklen vermeiden -> ExpressionChangedAfterItHasBeenChecked error, infinite circular updates oder andere auto updates können schwer nachvollziehbar werden

Effects sind an den **aktuellen context eines components gebunden.** Dh wird des component recycelt dann wird auch der effect recycled. Kann man aber natürlich manuell ( .destroy() ) machen.

# Was ist der Unterschied zwischen Signals und Observables?

**Signals sind reactiv** dh sie sind jederzeit abrufbar und verfügen jederzeit über einen value. Bei einem Observable muss man sich zuerst auf das **Observable subscriben** und kann die value dann lesen. **Observables versenden die Objekte über Zeit.** Dh erst wenn man sich subscribed und ein update passiert, wird es sichtbar. Heißt **signals sind pull-based**, können **jederzeit abgerufen** werden und **Observables sind push based**.

# Forms

Es gibt **2 Grundtypen** von Forms.
- Reactive forms
- Template driven forms

Man sollte **vor dem Einsatz entscheiden** welchen Typ man nimmt. Grundsätzliche werden beide folgend unterschieden:

1. Reactive Forms

Bieten **direkten Zugriff auf das Formelement.** Heißt diese Form ist leicht wiederverwendbar und testbar. Grundsätzlich soll man diesen Ansatz wählen, wenn die Applikation fast ausschließlich komplizierte Forms beinhaltet

2. Template Driven Forms

Werden mittels einer **directive** an das Model gebunden. Heißt mittels **[ngModel].**
Wird verwendet für sehr einfach Forms wie zb. **Newsletter subscribe oder normaler Login**.

Bei einer Form kann man folgenden Type's verwenden:
- **FormControl**: bindet das Steuerelement an ein Formelement und trackt Validierung und value
- **FormGroup**: beinhaltet mehrere FormControls
- **FormArray**: trackt values von mehreren FormControls in einem Array
- **ControlValueAccessor**: wird verwendet um direkt auf DOM Elemente zuzugreifen