

# Formulare

- Umsetzbar auf zwei Wege
  - Vorlagen-getrieben
    - Dabei gibt die Vorlage das Formularmodel und die Validatoren vor (ähnlich AngularJS)
  - Reaktiv (Daten-getrieben)
    - Hierbei werden die Formularelemente vorab geplant und an ein Formular in der Vorlage gebunden

# Formulare - Vorlagen-getrieben

- Vorbereitend: Einbindung des **FormsModuls** zur
- Anschließend sind Formular-Direktiven in der Vorlagen-Schicht nutzbar:
  - **ngModel, required, minlength, ...**
  - zur Bindung von Validatoren und Werten ins Formular-Model
  - All dies wird ohne zusätzliche Programmierung realisiert

# Formulare - Vorlagen-getrieben

- **ngForm** – wird genutzt, um das Formular auszuzeichnen.
- Direktive verfügt über ein **exportAs** d.h. wir können dies für einen #Hash-Id zuordnen **#myForm='ngForm'**
  - Ermöglicht den Zugriff auf Control-Eigenschaften
    - **valid, invalid, value** etc.
    - **myForm.valid**

# Formulare - Vorlagen-getrieben

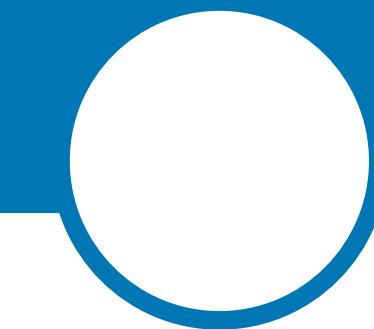
- **ngModel** kann auf drei Arten genutzt werden
  - Als Attributs-Direktive **ngModel** kombiniert mit einer Namensdefinition über das **name** Attribut.
    - Dadurch wird automatisch ein Formular-Model erzeugt
    - **myForm.value = {name: Input-Feld-Wert}**
  - Als Attributs-Direktive mit Bindung eines Initial-Wertes **[ngModel]**

# Formulare - Vorlagen-getrieben

- Vermeide: Nutzung als Attributs-Direktive mit Zweiwege-Bindung `[(ngModel)]`. Dadurch wird der Initial-Werte aktualisiert. D.h. es gibt zwei Modelle ☹
- Als Zuweisung für eine `#Hash-Id` z.B. `#mail='ngModel'`
  - Ermöglicht kombiniert mit der `ngModel` Direktive den Zugriff auf: `valid`, `invalid`, `value` etc.
  - `mail.valid`

# Formulare - Vorlagen-getrieben

- **ngModelGroup** Direktive zur Gruppierung von Model-Informationen
  - Die Direktive muss hierarchisch in der Vorlage genutzt werden.
  - Die **input**-Knoten des Direktiven-Elementes erzeugen die Gruppenelemente.



## Form

```
<form novalidate #myForm="ngForm">  
  <input type="text"  
    autocomplete="name"  
    placeholder="name"  
    name="name" _  
    #name="ngModel"  
    ngModel  
  >  
  <span ngModelGroup="credentials">  
    <input name="email"  
      #email="ngModel" ngModel>  
    <input name="password"  
      #password="ngModel" ngModel>  
  </span>  
</form>
```



## Model

ngForm -> myForm  
ngModel -> name  
ngModelGroup -> credentials  
ngModel -> email  
ngModel -> password

## Form

```
<form novalidate #myForm="ngForm">  
  <input type="text"  
    autocomplete="name"  
    placeholder="name"  
    name="name" ——————  
    #name="ngModel"  
    ngModel  
  >  
  <span ngModelGroup="credentials">  
    <input name="email"  
      #email="ngModel" ngModel>  
    <input name="password"  
      #password="ngModel" ngModel>  
  </span>  
</form>
```

## Model

```
myForm.value = {  
  name: '...',  
  credentials {  
    email: '...',  
    password: '...',  
  }  
}
```

- Erzeuge ein Formular, dass folgendes Model abbildet:

```
{ name: '...',  
  credentials: {  
    email: '...',  
    password: '...'  
  } }
```

Übung 17

# Formulare – Controls-

- **ngForm** und **ngModel** – sind Control-Direktiven mit folgenden Eigenschaften:
  - **value** - Wert
  - **valid, invalid** - Valide
  - **touched, untouched** - Berührt
  - **dirty, pristine** – Benutzt/Unbenutzt
  - **errors?** – Validator-Fehler

# Formulare – Controls

- Control Methoden:
  - `setValue, reset` – Wert
  - `markAsTouched, markAsUntouched` - Berührt
  - `markAsDirty, markAsPristine` – Benutzt/Unbenutzt
  - `setErrors?` – Validator-Fehler

# Formulare – Validatoren

- Validatoren lassen sich über Direktiven einbinden
  - **required** – erforderlicher Wert
  - **email** – Gültige Mail
  - **minlength, maxlength** – Längen-Prüfung
  - **pattern** – Ausdrucks-Prüfung

# Formulare – Validatoren

- Validatoren legen im `errors` Objekt des Controls Fehlerinformationen in abh. zum Validator ab.
- Fehlermeldungen lassen sich entsprechend darstellen
- `<div *ngIf="email.errors?.required">...</div>`
  - Das Fragezeichen bindet optionale Werte

# Formulare – Daten senden

- **(submit)** – Verwenden auf dem Formular das Submit-Ereignis
- Nutzen als Auslöser im Formular einen **<button>** oder **<a>** vom Typ **submit**
  - Verwende auf dem Auslöser zusätzlich die **disable**-Direktiven, zum Deaktivieren bei ungültigen Formularen.

```
<form novalidate #myForm="ngForm" (submit)="send(myForm)">
```

```
<button type="submit" [disabled]="myForm.invalid">senden</button>
```

# Formulare – Daten zurücksetzen

- (reset) – Verwenden auf dem Formular das Reset-Ereignis
  - Nutzen als Auslöser im Formular einen <button> oder <a> vom Typ reset
  - Verwende auf dem Auslöser zusätzlich die disable-Direktiven, zum deaktivieren, wenn noch keine Formularwerte eingetragen sind Formularen.

```
<form novalidate #myForm="ngForm" (submit)="send( myForm )"  
      (reset)="reset( myForm, $event )">
```

```
<button type="reset" [disabled]="!myForm.dirty">reset</button>
```

- Erweitere das Formular aus Übung 17
- Verwenden Validatoren und Zeige Fehler an
- Nutze das submit Ereignis und gebe in der Console die Werte aus
- Nutze das reset Ereignis und setze die Werte zurück auf die initialWerte. Verwendet dafür die setValue Methode des Controls

Übung 18

# Formular CSS-Klassen

- Angular fügt an input-Elemente autom. CSS-Klassen, die den Status des Controls wiederspiegeln.
  - ng-untouched, ng-touched
  - ng-pristine, ng-dirty
  - ng-invalid, ng-valid

# Model-Optionen

- Die gleichnamige Direktive beeinflusst das Model-Handling
  - [ngModelOptions]="{name: 'name'}"
    - ersetzt das setzen des name-Attributes
  - [ngModelOptions]="{standalone: true}"
    - Wert wird dem übergeordneten Form nicht mitgeteilt

# Model-Optionen

- `[ngModelOptions]="{updateOn : 'blur '}"`
  - Definiert einen Form-Hook (`change`, `submit`, `blur`) bei dem das Model aktualisiert werden soll.
  - `debounce` - angekündigt: Update nach timeout.

# Eigene Validatoren

- Validatoren bestehen aus zwei Schichten
  - ValidatorFn
    - Funktion übernimmt die Überprüfung und reicht aus, wenn die Nutzung ausschließlich reaktiv ist.
  - Direktive
    - Nutzt die zuvor beschrieben Funktion und ist für vorlagengetriebene Formulare notwendig

# Validator-Funktion

- Funktion wird über eine Factory erzeugt, welche optional die Prüfungsbedingung entgegennimmt.
- **ValidatorFn** - erwartet: **AbstractControl**
  - gibt ein Fehlerobjekt (**ValidationErrors**) oder null zurück

# Validator-Funktion

```
export class EqualValidator {  
    static isEqual ( compare: any ): ValidatorFn {  
        return ( control: AbstractControl ): ValidationErrors | null => {  
            if ( control.value === null || compare === null ) return null;  
            return compare !== control.value ?  
                { 'equal': { 'is': control.value, 'should': compare } } : null;};  
    }  
}
```

# Validator-Direktive

- Validierung wird in `NG_VALIDATORS` durch eine neue Direktive erweitert
- Erweiterung wird im Injector der Direktive bereitgestellt.
- Direktive muss das `Validator` Interface implementieren.
  - `validate ( c: AbstractControl ): ValidationErrors | null`
    - Wird zur Prüfung ausgeführt und gibt Fehlerobjekt zurück

# Validator-Direktive

- `registerOnValidatorChange (fn: any): void;`
- Methoden übermitteln eine Referenz zu den, `onChange` Callback
  - Notwendig, wenn Prüfungsbedingungen sich ändern

# Validator-Direktive – Injector erweitern

- Erweitere den `NG_VALIDATORS`
- Benutze hierfür den `ExistingProvider`
- Als Provide-Token nutze: `NG_VALIDATORS`
- Der Wert wird über `useExisting` auf die Direktiven-Klasse gesetzt, da diese nicht unmittelbar im Injector existiert wird die `forwardRef` Methode genutzt
- `multi: true` erweitert die `NG_VALIDATORS` Liste

# Validator-Direktive – Injector

- ```
export const EQUAL_VALIDATOR = {  
  provide: NG_VALIDATORS, multi: true,  
  useExisting: forwardRef(() => MyDirective) };
```
- ```
@Directive ( { selector: ,[equalValidator][ngModel]',  
  providers: [EQUAL_VALIDATOR] } )
```
- ```
export class MyDirective implements Validator {
```

- Erstelle auf Basis des equal Validators einen Zukunfts-Validator
- Dieser soll prüfen, ob ein gewähltes Datum in der Zukunft liegt.
- Nutzbar soll der Validator nur auf `input[type=„date“]` sein

Übung 18

# Werte Zugriff

- Falls Werte manipuliert werden müssen, bevor sie im Model oder der View genutzt werden
- Zugriff-Steuerung: `NG_VALUE_ACCESSOR` durch neue Direktive erweitern
- Erweiterung im Injector der Direktive bereitstellen.
- Direktive muss das `ControlValueAccessor` Interface implementieren.

# Werte Zugriff – Interface Methoden

- `writeValue(obj: any): void;`
  - Aufgerufen bei Modeländerungen von Form-API. Methode muss View anpassen.
- `registerOnChange & registerOnTouched (fn: any): void;`
  - Methoden übermitteln eine Referenz zu den, onChange und onTouched Callback

# Werte Zugriff – Interface Methoden

- **onChange** (value): void;
  - Callback wird aufgerufen beim Change-Ereignis
  - Übermittel wird der aktuelle Wert aus der UI.
  - Callback aktualisiert das Model über die Form-API

# Werte Zugriff – Interface Methoden

- `onTouched (): void;`
- Callback aufrufen, wenn der Status des Controls geändert werden muss
- Status gibt wieder, ob das Formular-Element aktiviert(focus/blur reicht) wurde.

# Werte Zugriff – Injector erweitern

- Erweitere den `NG_VALUE_ACCESSOR`
- Benutze hierfür den `ExistingProvider`
- Als Provide-Token nutze: `NG_VALUE_ACCESSOR`
- Der Wert wird über `useExisting` auf die Direktiven-Klasse gesetzt, da diese nicht unmittelbar im Injector existiert wird die `forwardRef` Methode genutzt
- `multi: true` erweitert die `NG_VALUE_ACCESSOR` Liste

# Werte Zugriff – Injector erweitern

- ```
export const CONTROL_VALUE_ACCESSOR = {  
  name: 'formatterParserValueAccessor',  
  provide: NG_VALUE_ACCESSOR, multi: true,  
  useExisting: forwardRef(() => MyDirective) };
```
- ```
@Directive ( { selector: 'input[msgFormater]',  
  providers: [CONTROL_VALUE_ACCESSOR] } )
```
- ```
export class MyDirective implements ControlValueAccessor {
```

- Beispiel erläutere das Grundprinzip anhand der Demo 1

Demo 1

# Reaktive Formulare

- Im Gegensatz zu Vorlagen-getriebenen Formularen vermeiden wir Direktiven wie: `ngModel`, `required`, `minlength`
- Statt dessen werden zuvor Controls erzeugt und anschließend in der Vorlage gebunden via:
  - `formGroup`, `FormControl`, `FormControlName` ...
  - Als Vorbereitung muss das `ReactiveFormsModule` eingebunden werden.

# Reaktive Formulare – Model erzeugen

- Erzeuge Controls für Werten über **FormControl**
  - Konstruktor erwartet **Wert** und **Validatoren**
- Erzeuge Gruppen von Werten über **FormGroup**
  - Konstruktor erwartet ein **Key-Value-Pair Objekt**
    - **Key:** Name des Controls oder der Untergruppe
    - **Value:** Instanz des Controls oder der Untergruppe

# Reaktive Formulare – Direktiven

- **[formGroup]** – Bindet die unterste Wert-Gruppe
- **formGroupName** – Bindet Untergruppe anhand des Names, das im Key-Value-Pair Objekt definiert wurde.
- **formControlName** – Bindet Control anhand des Names, das im Key-Value-Pair Objekt definiert wurde.
- **[formControl]** – Bindet eine Control-Instanz.

# Form

```
<form novalidate [FormGroup]="myForm">  
  <input type="text"  
    formControlName="name"  
  >  
  <span formGroupName="credentials">  
    <input type="email"  
      formControlName="email">  
    <input type="password"  
      formControlName="password">  
  </span>  
</form>
```

# Model

```
this.myForm = new FormGroup ( {  
  name: new FormControl ( 'Saban',  
    Validators.required ),  
  credentials: new FormGroup ( {  
    email : new FormControl (  
      'us@netTrek.de',  
      [ Validators.email,  
        Validators.required ] ),  
    password: new FormControl ( ... ) } );
```

# Form

```
<form novalidate [FormGroup]="myForm">  
  <input type="text"  
    formControlName="name"  
  >  
  <span formGroupName="credentials">  
    <input type="email"  
      formControlName="email"  
    <input type="password"  
      formControlName="password">  
  </span>  
</form>
```

# Model

```
myForm.value = {  
  name: '...',  
  credentials {  
    email: '...',  
    password: '...',  
  }  
}
```

- Erzeuge ein Reaktives-Formular, dass folgendes Model abbildet:

```
{ name: '...',  
  credentials: {  
    email: '...',  
    password: '...'  
  } }
```

Übung 19

# Formulare – Helfer – FormBuilder

- FormBuilder (DI) –Service vereinfacht die Model Erstellung und den Umgang mit FormControl und FormGroup
  - Anstelle von new FormGroup () nutzen wir die group Methode vom FormBuilder und übergeben ein Key-Value Objekt.
  - Key: Name des Controls oder der Untergruppe
  - Value: Eigenschafts-Array oder Untergruppe via group Methode

# Formulare – Helfer – FormBuilder

- **Value**: Eigenschafts-Array
  - Erstes Element – Startwert
  - Zweites Element: Validator oder Validator-Array

# Formulare – Helper – FormBuilder

```
this.myForm = this.fb.group({  
    name: [ 'Saban', Validators.required ],  
    credentials: this.fb.group ( {  
        email: ['us@netTrek.de', [ Validators.email,  
            Validators.required ]],  
        password: ['test1234', Validators.required ]  
    })  
});
```

# Formulare – Helfer – Control

- **get**: Methode gibt ein Control aus dem Model zurück
  - Parameter:
    - Name des Controls
    - oder Pfad (Names-Array) zu einem Control
  - `this.myForm.get( ['credentials', 'email'] ) as FormControl;`

# Formulare – Helfer – Control - Fehler

- **hasError** : Methode gibt ein Boolean zurück, ob ein bestimmter Validator-Fehler existiert
- Parameter:
  - Name des Errors z.B. **required**, **email** ...
  - Name des Controls oder Pfad (Names-Array) zu einem Control

# Formulare – Helfer – Control - Werte

- `setValue( value: any, opts?): void;`
  - `onlySelf?` : boolean [default: false]
    - Validation nur auf Control nicht auf Eltern-Komponente
  - `emitEvent?` : boolean [default: true]
    - `valueChanges` Event wird vom Control gefeuert

# Formulare – Helfer – Control - Werte

- `setValue( value: any, opts?): void;`
- `emitModelToViewChange?: boolean`
  - View wird via `onChange` über die Änderung informiert
- `emitViewToModelChange?: boolean`
  - Model wird via `ngModelChange` über die Änderung informiert

# Formulare – Helfer – Control - Werte

- `reset( value, opts?: { onlySelf?: boolean;  
emitEvent?: boolean; })`
- Setzt Control zurück
  - `value` = null oder Wert
  - Zustand wird auf `pristine` & `untouched` gesetzt

# Formulare – Helper – Control - Status

- `markAsTouched( opts?: { onlySelf?: boolean; }): void;`
- `markAsUntouched( opts?: { onlySelf?: boolean; }): void;`
- `markAsDirty( opts?: { onlySelf?: boolean; }): void;`
- `markAsPristine( opts?: { onlySelf?: boolean; }): void;`
- `disable(opts?: { onlySelf?: boolean; emitEvent?: boolean; })`
- `enable (opts?: { onlySelf?: boolean; emitEvent?: boolean; })`