TEMPLATE-DRIVEN VS. MODEL-DRIVEN

# Reine Formsache

Angular bietet zwei Verfahren für das Arbeiten mit HTML-Formularen an.

Webanwendungen auf Basis von JavaScript/TypeScript und HTML5 sind moderner denn je. Viele Unternehmen wählen nach eingehender Validierung der Technologien diese beiden, um ihre Oberfläche umzusetzen. In diesem Zusammenhang treffen die Verantwortlichen häufig auf das Web-Framework Angular. Es bietet eine komfortable Plattform zum Erstellen von einfachen Single-Page-Applikationen bis hin zum Umsetzen schwergewichtiger Geschäftsanwendungen, die komplexe Prozesse abbilden.

Dabei stehen sich die Ansätze von „Template-driven" (vorlagenbasierten) und „Model-driven" (modellbasierten) Anwendungen mit ihrer jeweiligen Art der Umsetzung und ihren Vor- und Nachteilen gegenüber, hier gezeigt unter Verwendung von Angular am Beispiel von Datenbindung und Validierung. Auch die Herausforderungen, die sich bei der Wahl eines Ansatzes stellen und wie diese zu bewältigen sind, kommen zu Wort.

## Warum Angular Forms?

Viele Anwendungen scheinen nur aus Formularen zu bestehen. Dabei muss jede Applikation die gleichen Probleme bewältigen: Sie muss Daten erst auswerten und danach speichern. Dabei reicht die Validierungslogik von einfachen Regeln, etwa dass ein Feld nicht leer sein darf, bis zu komplexen Geschäftsregeln, dass etwa Datenabfragen an das Backend gestellt werden müssen. In der HTML5-Spezifikation gibt es dafür das Constraint Validation API [1], mit dem Felder auf verschiedene Logik geprüft werden können.

Wer Angular verwendet, genießt den Vorteil, nicht nur ein, sondern sogar zwei Verfahren (und entsprechende Module) zum Verwalten von Formularen verwenden zu können: die Ansätze für vorlagenbasierte und modellbasierte Formulare. Sie sind dafür ausgelegt, ein über Datenbindung verknüpftes Datenmodell zu validieren, und können im Fehlerfall Validierungsfehler anzeigen.

## Template- oder Model-driven

Es ist sinnvoll, sich im Vorfeld für eine der beiden Varianten zu entscheiden, um die jeweilige Anwendung in dieser Hinsicht einheitlich und konsistent zu halten. Mit Blick auf den Lebenszyklus der Software ist dann die Lesbarkeit des erzeugten Codes gewährleistet und zu einem späteren Zeitpunkt kann sich ein Entwickler schnell in Aufbau und Logik einarbeiten. Dies begünstigt zugleich die Wartbarkeit, da durch die Wahl einer Variante bei Anpassungen Validierungslogik entweder im HTML- oder im TypeScript-Code zu finden ist.

Kurz gesagt wird die zum Formular gehörige Logik beim vorlagenbasierten Ansatz im HTML-Code eingebaut und beim modellbasierten in der TypeScript-Datei des Controllers. Im Folgenden werden die Unterschiede zwischen beiden anhand von Beispielen für die Datenbindung und die Validierung dargestellt. Um selbst Hand anzulegen empfiehlt es sich, eine leere Angular-Anwendung mithilfe des Angular-CLI (Command Line Interface) [2] zu erzeugen, da auf diese Weise schnell und einfach das Grundgerüst angelegt ist und die Formulare schnell nachgebaut sind.

## Binding

Geschäftsanwendungen sind häufig darauf ausgelegt, den Nutzer über unterschiedliche Masken und Formulare zu führen, in denen jeweils eine gewisse Anzahl von Feldern auszufüllen ist. Dies kann vom Anlegen einfacher Nutzerdaten bis hin zur Pflege von Stamm- oder Bewegungsdaten reichen,

**Listing 1: Die Datei template-form.template.html für ein vorlagenbasiertes Formular**

```
1: <form #registrationForm="ngForm" (ngSubmit)="onSubmit(registrationForm.value)">
2:   <input type="text" [(ngModel)]="user.emailAddress" name="emailAddress" required email
       #emailAddress="ngModel">
3:   <div [hidden]="!emailAddress.errors?.email || emailAddress.pristine">The email address is not correct!
       </div>
4:   <input type="password" [(ngModel)]="user.password" name="password" required #password="ngModel">
5:   <input type="password" [(ngModel)]="user.confirmationPassword" name="confirmationPassword"
       required #confirmationPassword="ngModel">
6:   <button type="submit" [disabled]="!registrationForm.valid">Submit</button>
7: </form>
```

**Listing 2: Die Datei template-form.component.ts**

```
1: import { Component } from "@angular/core";
2: import { User } from "./../user.interface";
3: @Component({
4:   selector: "app-template-form",
5:   templateUrl: "./template-form.template.html"
6: })
7: export class TemplateFormComponent {
8:   public user: User;
9:   constructor() { this.user = {} as User; }
10:   private onSubmit(model: User) {
11:   { console.log(model); }
12: }
```

wobei Regeln zur Eingabe berücksichtigt werden müssen. Angular bietet mit seinem Forms-Mechanismus [3] eine komfortable Möglichkeit, solche Oberflächen zu gestalten.

Aus AngularJS ist die Direktive *ng-model* bereits bekannt; sie stellt eine zweiseitige Datenbindung zwischen dem Modell und den Daten eines Formulars her. Angular 2 bietet die gleichnamige Direktive *ngModel* an, die eine identische Funktionalität mitbringt. Verfahren, die mithilfe dieser Direktive verwirklicht werden, werden als „Template-driven Forms" (vorlagenbasierte Formulare) bezeichnet.

Um in der Anwendung die *ngModel*-Direktive für Template-driven Forms zu nutzen, ist zunächst das entsprechende Modul (*FormsModule*) in die Datei *app.modules.ts* einzubinden:

```
import { FormsModule, ReactiveFormsModule }
  from "@angular/forms";
  ...
  @NgModule({
    declarations: [
     ...
    ],
    imports: [
      ...,
      FormsModule,
      ReactiveFormsModule,
      ...
    ],
    ...
})
```

Es fällt auf, dass hier auch das Modul *ReactiveFormsModule* eingebunden wird. Dadurch stehen auch die Formulare des modellbasierten Ansatzes zur Verfügung und beide Ansätze können auch parallel verwendet werden.

Nun lässt sich auch schon gleich die erste Oberfläche erstellen, in der eine E-Mail-Adresse und ein Passwort anzugeben sind; das Passwort soll zur Sicherheit ein weiteres Mal eingegeben werden.

Das dazugehörige Template ist in Listing 1 zu sehen. In Zeile 1 wird allgemein mit dem HTML-Tag *<form>* das Formular aufgespannt, wobei zusätzlich das spezifische Attribut *#registrationForm="ngForm"* angegeben ist, das auf die *ngForm*-Direktive verweist. Diese stellt in dem Tag Funktionen bereit wie die Übernahme der Eingaben in den mit den Attributen *name* und *ngModel* versehenen *<input>*-Elementen als Eigenschaften (siehe Listing 1, Zeile 2). Mit dem Attribut *(ngSubmit)* im *<form>*-Tag verknüpft Angular das Formular mit der Methode, welche die im Formular eingetragenen Daten zum Server überträgt. Der Inhalt des Formulars lässt sich gleichzeitig als Input-Parameter mitgeben, falls man sich nicht durchgängig für eine zweiseitige Datenbindung entscheidet; so ist der Zugriff auf die Daten innerhalb der Submit-Methode möglich.

Die Direktive *ngModel* steht für das Binding zwischen *<input>*-Element und dem Modell im TypeScript-Code. Dabei gibt es drei Varianten, die innerhalb des Template-Ansatzes verwendet werden können, und unterschiedliche syntaktische Ausdrücke für die Datenbindung:

- **[(ngModel)]:** Es wird ein zweiseitiges Binding erstellt (in Listing 1 in den Zeilen 2, 4 und 5). Sobald sich der Wert im Eingabefeld ändert, wird der Wert im Modell von Angular angepasst. Zugleich stellt Angular den globalen Status der Eingabemaske des *<form>*-Tags bereit (valid/invalid), was für die Ausgabe von Benachrichtigungen praktisch ist.
- **[ngModel]:** Hierbei wird lediglich eine einfache Datenbindung eingerichtet. Eingabefelder werden zwar initialisiert, Änderungen ihres Inhalts schlagen sich nicht im Modell nieder. Diese stehen erst nach dem Submit innerhalb der Methode bereit und können dann verarbeitet werden. So lässt sich das Formular mit Daten, beispielsweise von einem Webdienst, im Vorfeld initialisieren und validieren.
- **ngModel:** Die dritte Variante wird lediglich innerhalb des *<input>*-Tags ohne Zuweisung eingesetzt und lässt so lediglich die Validierung durch Angular zu. Zu beachten ist außerdem, dass hier kein Binding zustande kommt. Diese Variante eignet sich beispielsweise für rein informative Eingaben.

Listing 2 zeigt den Code zur HTML-Vorlage, der lediglich das Interface für den zu erstellenden Nutzer in Zeile 2 importiert und die entsprechende Submit-Methode in Zeile 10 mit dem Konsolen-Output implementiert. Im Vergleich zum vorlagenbasierten Ansatz ist hier das Template und in Listing 3 der TypeScript-Code für den modellbasierten Ansatz abgebildet. Es fällt sofort auf, dass sich das Template in Listing 3 übersichtlicher liest, zugleich aber der Code länger ausfällt und weitere Module eingebunden werden müssen (siehe Zeile 1).

Beim modellbasierten Ansatz wird ebenfalls das *<form>*-Tag um den Zusatz *[FormGroup]* [4] erweitert (Listing 4 in Zeile 1). Dieser Zusatz bezieht sich auf die Eigenschaft des Controllers *form* in Listing 3 (Zeile 8) und bindet diese sogleich. Über die Angabe *[FormGroup]* und der darin angegebenen Eigenschaften kann mit dem Attribut *formControlName* (Listing 4, Zeilen 2, 4 und 5) in den *<input>*-Feldern auf diese verwiesen werden. So stellt Angular ohne Umwege eine zweiseitige Bindung bereit. Dahinter stehen lediglich mehrere ▶

*FormControls* [5] oder eine *FormGroup* wie in Listing 3 in Zeile 8, die sich aus den benötigten Controls zusammensetzt und die Funktionalität zur Validierung der gesamten Gruppe oder einzelner Eigenschaften bereitstellt. Wie beim vorlagenbasierten Ansatz können auch hier die Felder durch Angabe des Werts an erster Stelle des Arrays vorinitialisiert werden, siehe in Listing 3 die Zeilen 11, 14 und 15.

Grundsätzlich lässt sich auch die *ngModel*-Direktive verwenden. Davon ist aber abzuraten, da Angular den Inhalt des Formulars im Modell und zugleich in der *FormGroup* bereitstellen würde, was beim Entwickeln verwirren kann.

Aber warum sollte man auf *ngModel* verzichten und für das Formular eine *FormGroup* nachbilden? Weil es einfacher ist, Eigenschaften des Formulars im Blick zu behalten, ohne lange im HTML-Code danach suchen zu müssen. Zugleich ändert sich beim Hinzufügen von Feldern und Funktionalität das HTML beim modellbasierten Ansatz nur marginal, was es erleichtert, sich in den Code einzulesen und ihn gegebenenfalls anzupassen. Ferner stellt sich nicht die Frage, welche Art von Binding nötig ist, da Angular sie durch das Erweitern der *FormGroup* quasi implizit beantwortet.

## Validierung

In Listing 1 fällt auf, dass in Zeile 2 zwei weitere Attribute verwendet werden. In Listing 4 sind diese dagegen ausgespart. Hierbei handelt es sich um Mechanismen von Angular, um Eingaben des Formulars zu validieren.

In Zeile 2 von Listing 1 ist das Attribut *required* enthalten, das den Inhalt als erforderlich markiert und sich auf den globalen Zustand des Formulars auswirkt. Somit kann die Schaltfläche *<button>* mit *registrationForm.valid* für das Absenden aktiviert werden (Listing 1, Zeile 6).

Vergleichbar damit ist der modellbasierte Ansatz: Er verzichtet auf die Attribute im HTML und definiert die Validierung in der *FormGroup* (Listing 3, Zeile 11). Nach der Angabe des Anfangswerts können an zweiter Stelle innerhalb des Arrays benötigte Validatoren angegeben werden. Hier wurden beispielsweise der Angular-Pattern-Validator für die korrekte E-Mail-Adresse (*Validators.pattern([regex])*) und der *required*-Validator für eine obligatorische Eingabe verwendet.

Zurück zu Listing 1, Zeile 2: Hier findet sich ein weiteres Attribut, das sich auf eine selbst implementierte Validator-Direktive bezieht. Diese wird durch Angular selbstständig ausgewertet und gleichzeitig in Zeile 3 durch *emailAddress.errors?.email* abgegriffen. Der ?-Operator bewahrt Angular bei nicht gesetztem Attribut vor Fehlermeldungen. Mehr zur Implementierung eigener Validatoren gibt es in der Angular-Dokumentation zur Gültigkeitsprüfung von Formularen [6].

Um die Eingabe identischer Passwörter zu prüfen, setzt der vorlagenbasierte Ansatz wieder auf eine Darstellung innerhalb HTML. So lässt sich das HTML durch eine entsprechende Fehlermeldung erweitern, die sich auf die eingegebenen Passwort-Werte bezieht, wie hier in der ersten Zeile:

```
<div [hidden]="(user.confirmationPassword ===
  user.password) || (confirmationPassword.pristine ||
  password.pristine)">The passwords do not match!</div>
<button type="submit" [disabled]="
  !registrationForm.valid || (user.confirmationPassword
  !== user.password)">Submit</button>
```

Der modellbasierte Ansatz hingegen lässt es zu, die Validierung innerhalb oder außerhalb des Controllers zu implementieren (Listing 3, Zeile 20). Die Validatoren werden an die entsprechende Sub-*FormGroup* (Zeile 13 bis 17) gehängt und so bei jeder Änderung oder Eingabe der Felder geprüft.

Dies ermöglicht es, die Validatoren über verschiedene Oberflächen hinweg mit identischen Anforderungen zur Verfügung zu stellen und wiederzuverwenden. Das Auslagern der Validatoren vom HTML in den Controller macht die Templates schlanker und besser lesbar. Zugleich muss die Prüflogik im Controller abgebildet werden, was zunächst zwar ei-

**Listing 3: Die Datei model-form.component.ts für ein modellbasiertes Formular**

```
1: import { AbstractControl, FormBuilder, FormControl,
2:   FormGroup, ValidatorFn, Validators }
3:   from "@angular/forms";
4: import * as _ from "lodash";
5: @Component({ selector: "app-model-form",
6:   templateUrl: "./model-form.template.html", })
7: export class ModelFormComponent {
8:  public form: FormGroup;
9:  constructor(formBuilder: FormBuilder) {
10:    this.form = formBuilder.group({
11:      emailAddress: ["", [Validators.required,
12:        Validators.pattern(".+@.+\..+")]],
13:      passwords: formBuilder.group(
14:        { password: ["", Validators.required],
15:        confirmationPassword:
16:          ["", Validators.required] },
17:      { validator: this.match }),
18:    });
19:  }
20:  private match(c: FormGroup) {
21:    var inputs = [];
22:    _.each(c.controls, (control) =>
23:    { if (!_.includes(inputs, control.value))
24:      { inputs.push(control.value); } });
25:     return inputs.length === 1
26:       ? true: { match: { valid: false } };
27:   }
28:   private onSubmit()
29:   { console.log(this.form.value); }
30: }
```

**Listing 4: Die Datei model-form.template.html für ein modellbasiertes Formular**

```
1: <form [formGroup]="form" (ngSubmit)="onSubmit()">
2:     <input type="text" formControlName="emailAddress">
3:     <div [hidden]="form.controls.emailAddress.valid || form.controls.emailAddress.pristine">
         The email address is not correct!</div>
4:     <input type="password" formControlName="password">
5:     <input type="password" formControlName="confirmationPassword">
6:     <button type="submit" [disabled]="!form.valid">Submit</button>
7: </form>
```

nen Mehraufwand mit sich bringt, aber vor Duplikaten über mehrere Formulare hinweg schützt, wenn die Validatoren generisch implementiert werden und global zugänglich sind.

## Pro und Kontra

Der vorlagenbasierte Formularansatz ähnelt dem Ansatz von AngularJS. Wer also eine AngularJS-Applikation migriert, wird es einfacher haben; viele Views können bestehen bleiben, ohne sie anpassen zu müssen, und die Formularsyntax kann übernommen werden. Außerdem ist die Anwendung von Formularen beim Template-driven-Ansatz einfacher und bei simplen Szenarien vorzuziehen.

Wer jedoch eine komplexe Anwendung entwickelt, wird sich nach dem höheren Einarbeitungsaufwand mit dem Model-driven-Ansatz leichter tun. Hauptsächlich weil bei vorlagenbasierten Formularen vieles, wie der Name schon sagt, innerhalb der HTML-Vorlage definiert wird, während in Formularen des modellbasierten Ansatzes alles mit TypeScript beschrieben wird. Das macht auch das Testen einfacher. Wer Unit-Tests für seine Prüflogik verwenden will, kann bei modellbasierten Formularen das Modell im Hintergrund anlegen und prüfen, ob es gültig ist. Für den vorlagenbasierten Ansatz muss das Template als DOM-Element angelegt werden und lässt sich dann erst prüfen.

Ein weiterer Vorteil von modellbasierten Formularen ist, dass sich „reactive transformations" anwenden lassen. So ist es möglich, mit dem reaktiven Ansatz auf Änderungen oder Events zu reagieren.

Wer sich zudem mit Konzepten wie „Ahead of Time Compilation" und Angular Universal für das Rendern mit Angular auf dem Server beschäftigt, wird auch hier keine Unterschiede finden. Beide Varianten erlauben die Kompilierung auf dem Server.

## Fazit

Doch welcher Ansatz ist nun der geeignete? Wer von einem AngularJS-Projekt migriert oder ohne komplexe Validierung auskommt, wird sich mit dem vorlagenbasierten (Template-driven) Formularen leichter tun. Die Syntax ist ähnlich und erfordert wenig Aufwand, um bestehende Logik umzustellen.

Allen, die ein neues Angular-Projekt beginnen, sei zu modellbasierten (Model-driven) Formularen geraten. Sie ermöglichen es, komplexe Szenerien einfach umzusetzen. Und auch schwierige Fälle wie abhängige Felder oder das Verwalten von großen und mehreren Formularen sind möglich.

Am Ende ist es auch immer eine Geschmackssache. Wer seine Daten gerne innerhalb des HTML-Templates auswertet, statt TypeScript zu definieren, dem ist mit dem vorlagenbasierten Ansatz besser geholfen. Zwar ist das Mischen beider Varianten möglich, allerdings um den Preis einer übersichtlichen Struktur.

Wer sich nun nach der Gegenüberstellung der beiden Ansätze immer noch nicht entscheiden kann, der kann mit modellbasierten Formularen jedoch nichts falsch machen. ■

*[1] HTML5, Constraints, www.dotnetpro.de/SL1801Angular2Forms1*
*[2] Angular CLI, https://cli.angular.io*
*[3] Angular: Forms, www.dotnetpro.de/SL1801Angular2Forms2*
*[4] Angular: FormGroup, www.dotnetpro.de/SL1801Angular2Forms3*
*[5] Angular: FormControl, www.dotnetpro.de/SL1801Angular2Forms4*
*[6] Angular: Form Validation, www.dotnetpro.de/SL1801Angular2Forms5*

**Florian Bader**
ist Consultant im Bereich Microsoft .NET bei der AIT GmbH & Co. KG. Er berät Kunden zum Application Lifecycle Management mit Schwerpunkt auf Softwareentwicklung und -architektur, Build-automatisierung und Qualitätsmanagement.
**florian.bader@aitgmbh.de**

**Lukas Ochsenreiter**
ist Consultant bei AIT GmbH & Co. KG. Er ist als Softwareentwickler mit Schwerpunkt Webentwicklung im industriellen Umfeld tätig.
**Lukas.ochsenreiter@aitgmbh.de**

**dnpCode** A1801Angular2Forms