Host Europe

Angular

Neue Techniken für Web-Professionals

E-Book

Inhaltsverzeichnis

Der Redakteur

Oliver Lindberg

Oliver Lindberg ist ein unabhängiger Redakteur, Content-Consultant, und Gründer von Pixel Pioneers, einer Konferenz für Frontend-Entwickler und UX/UI Designer. Ehemals Chefredakteur der wegweisenden Zeitschrift 'net magazine', beschäftigt Oliver sich inzwischen seit mehr als 15 Jahren mit Webdesign und -entwicklung und hilft internationalen Unternehmen bei der Umsetzung von erfolgreichen Content(-Marketing)-Strategien.

Vorwort

Angular ist nach wie vor eines der beliebtesten JavaScript-Frameworks für die Entwicklung moderner Web Apps. Die Vorteile liegen in der Konsistenz, Produktivität, Wartbarkeit, Modularität und frühzeitigen Fehlererkennung – alles Gründe, warum sich Teams immer wieder für Angular entscheiden. Da das Framework ein All-in-One Paket bietet, eignet es sich besonders für komplexe Anwendungen.

In diesem praxisorientierten Ebook wollen wir eine Reihe neuer Techniken vorstellen, die das Arbeiten mit Angular verbessern und vereinfachen. Alle Artikel sind voller Tipps, Ressourcen und anschaulichen Code-Beispielen, damit Sie den Tutorials einfach folgen und das gelernte Wissen vertiefen bzw. auf Ihre eigenen Projekte anwenden können.

Als Autoren haben wir einige der besten Angular-Experten gewonnen, die es im deutschsprachigen Raum gibt. Alle sind erfahrene Trainer, die sich in der Community engagieren und auf öffentlichen sowie privaten Veranstaltungen ihre Kenntnisse teilen: Auf den folgenden Seiten beschäftigt sich Gregor Woiwode mit zentralem State-Management für Angular mit NgRx, Martina Kraus widmet sich Angulars neuer Rendering-Engine Ivy, und David Würfel nimmt sich das NestJS-Framework vor, und wie eine Projektstruktur mit Angular-Front- und NestJS-Backend aussehen kann. Natürlich dürfen Performance und Testing nicht fehlen. Daher zeigt uns Christian Liebel verschiedene Techniken, um selbst große Angular-Apps super-schnell auszuführen. Abschließend begeistert uns David Müllerchen für das End-to-End Testing von Webanwendungen mit Cypress.

Viel Spass beim Lesen!

Der Autor

Gregor Woiwode

Gregor Woiwode ist CTO der co-IT.eu GmbH. Er liebt die Entwicklung von Tools, die Programmierern erlauben, noch produktiver zu sein.
Als Sprecher, Trainer und Consultant unterrichtet er Techniken, um die Architektur von Angular Anwendungen kontinuierlich zu verbessern.
Für sein Engagement in der Community und durch seine Beiträge für verschiedene Open-Source-Software-Projekte wurde er im Mai 2019 als Google Developer Expert für Angular und Webtechnologien ausgezeichnet.
Außerdem geht er gerne Laufen und probiert sich von Zeit zu Zeit als Hobbykoch.

Zentrales State-Management für Angular

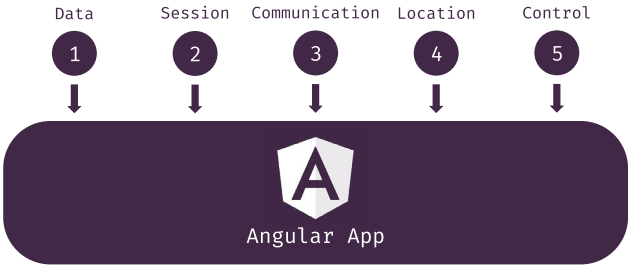
Warum ist es so herausfordernd, performante und bedienfreundliche Oberflächen zu bauen?

Eine Antwort lautet: Im Frontend müssen unterschiedliche Aufgaben gelöst werden. Es geht nicht nur um die Umsetzung von Logik. Bedienelemente müssen sinnhaft miteinander verknüpft werden. Das Design muss ansprechend gestaltet werden. Außerdem müssen die richtigen Daten zur richtigen Zeit visualisiert werden, um dem Nutzer ein komfortables Bedienerlebnis zu bieten.

Das höhere Ziel lautet: „Als Entwickler wollen wir eine Anwendung bieten, mit denen unsere Kunden ihr Business rocken können."

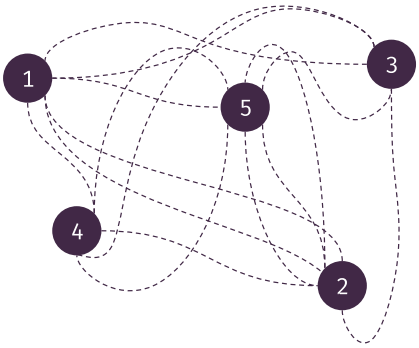
Das eben angesprochene Bedienerlebnis wird durch das Zusammenführen verschiedener Informationsquellen realisiert, die den aktuellen Anwendungsstatus repräsentieren. Im Frontend können fünf Statusarten unterschieden werden.

Die Komplexität der Fünf



Die fünf Arten des Applikationsstatus

Die Statusarten sind im Einzelnen im Artikel „The 5 Types of React Application State“ beschrieben. Wird jede Statusart einzeln betrachtet, ist die Komplexität vergleichsweise gering. Allerdings überschneiden sich die Status. Informationen werden aus *Location* (Bsp.: Query-Parameter) ausgelesen, um *Data* zu laden. Nicht jede *Session* hat das Recht, alles von *Data* abzurufen. Es entsteht ein Abhängigkeitsgraph. Dieser wächst mit jeder Funktion, die der Anwendung hinzugefügt wird.



Das Status-Spaghetti-Diagramm

Schnell sind Statusinformationen so miteinander gekoppelt, dass der betreffende Code monolithische Charakterzüge hat. Die Modularisierung und damit die Austauschbarkeit sinkt. Eine Änderung kann Fehler in anderen Teilen der Anwendung haben, die gar nicht abzusehen waren.
In Angular äußert sich dieser Zustand ziemlich deutlich. Eine entartete *Service-Composition* hält Einzug.

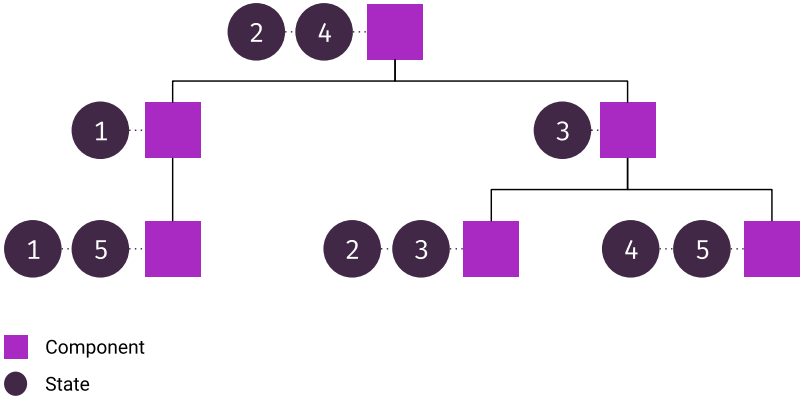
```
1  @Component({/*... */})
2  export class ListComponent {
3   constructor(
4     private route: ActivatedRoute,
5     private userService: UserService,
6     private shoppingService: ShoppingService,
7     private invoiceService: InvoiceService,
8   ) {}
9  }
```

Zum Code →

Service Composition in der Component

Services werden miteinander gekoppelt, sodass sie eigenständig nicht mehr funktionieren. Dass eine Änderung in *Service A* ein Fehlverhalten in *Service B* oder *C* hervorruft, ist eine große Gefahr. Bugs, die auf diese Art entstehen, führen nicht selten zu langen Debugging-Sessions, die Zeit und Nerven kosten.

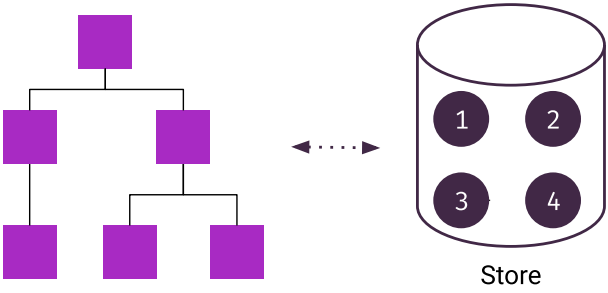
Außerdem zeigt das Code-Beispiel sehr gut, dass in einer Komponente verschiedene Status verarbeitet werden. Das heißt, dass in jeder Komponente *State-Management* betrieben wird. Die folgende Abbildung macht deutlich, wie *State* in einer komponentenbasierten Architektur verteilt ist.



Dezentralisiertes State-Management

Je größer der Component-Tree wird, desto herausfordernder ist es, die Zuständigkeiten richtig zu organisieren.

Das zentrale State-Management bietet für diese Problemstellung die Lösung. Zunächst werden die Statusarten in einem Punkt organisiert. Das ist vergleichbar mit einer In-Memory-Datenbank.



Zentralisiertes State-Management

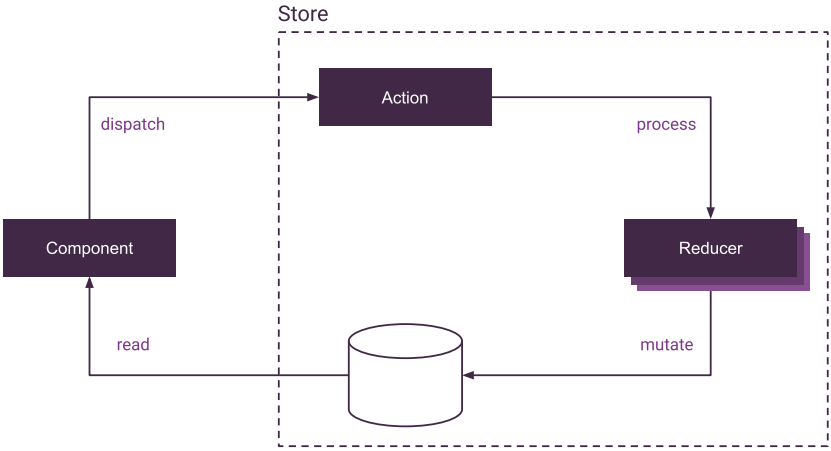
Des Weiteren werden alle Schreib- und Leseoperationen in der Anwendung in einer API homogenisiert. Wie genau das funktioniert, wird im nächsten Abschnitt am Beispiel der Redux-Architektur erklärt.

Was ist Redux?

Redux beschreibt den Datenfluss in einer Frontend-Anwendung. Dabei werden Lese- und Schreiboperationen voneinander getrennt (vgl. Command-Query-Segregation Pattern). Redux hat seine Wurzeln in Facebooks Flux-Architektur. Im Kern geht es darum, die Ereignisse in einer Anwendung besser zu organisieren. Ein Ereignis wird durch eine Interaktion mit der Benutzeroberfläche ausgelöst.

Redux-Logo

In Redux werden Ereignisse in eine ***Action*** verpackt und zu einem Store gesendet. Der Store ist eine Art In-Memory-Datenbank für Ihre Anwendung, in der der gesamte Anwendungszustand (vgl. Die Komplexität der Fünf) verwaltet wird. Dort werden die Informationen der ***Action*** ausgewertet und verarbeitet. Die folgende Grafik veranschaulicht den Datenfluss der Redux-Architektur.



Unidirektionaler Redux-Datenfluss

Das Schaubild kann mit wenigen Punkten erklärt werden:

1. Die ***Component*** versendet eine ***Action.***
2. Die ***Action*** wird durch s.g. ***Reducer***-Funktionen verarbeitet.
3. Der ***Store*** wird durch das Ergebnis der ***Reducer*** aktualisiert (mutiert).
4. Die ***Component*** kann Daten des ***Stores*** abonnieren und visualisieren.

In den folgenden Abschnitten zoomen wir in die einzelnen Bestandteile von Redux hinein, um zu verstehen, was sie im Einzelnen bedeuten.

Action

Die *Action* ist das Transportmedium, das eine Interaktion mit der Benutzeroberfläche repräsentiert. Darüber hinaus hat die *Action* noch eine weitere Aufgabe, die später erläutert wird. Eine *Action* wird durch ein serialisierbares JavaScript-Objekt repräsentiert.

```
1  const action = {
2    type: '[Counter] Add',
3    payload: 3
4  }
```

Zum Code →

Redux-Action

Jede *Action* muss über eine Property *type* verfügen. Optional kann sie eine Property *payload* besitzen. Es ist auch möglich, weitere Metadaten in das Objekt zu schreiben. Die *payload* kann jedweden Typ haben. Sie kann ein Primitive oder ein Referenztyp sein.

Immer wenn ein Benutzer auf die Oberfläche klickt, Eingaben tätigt oder Elemente per Drag & Drop verschiebt, wird die jeweilige Interaktion in eine *Action* verpackt und an den Store gesendet.

Die *Action* repräsentiert den Vertrag, über den in Redux kommuniziert wird.

Eine ***Action*** wird mithilfe des ***Store***-Services versendet, der in einer Komponente genutzt werden kann.

```
@Component({ /* ... */ })
export class CounterComponent {
   constructor(private store: Store<State>) {}

   add(count: number) {
       this.store.dispatch({
           type: '[Counter] Add',
           payload: count
       })
   }
}
```

Zum Code →

Versand einer Redux-Action

Nicht jede ***Component*** sollte mit dem ***Store*** kommunizieren. Hier gilt es die Zuständigkeiten klar abzugrenzen. Ein bewährtes Mittel ist die Trennung von Komponenten in Presentational- & Container-Components.

Store

Im *Store* werden *Actions* verarbeitet. Mithilfe von *type* und *payload* wird abgeleitet, welche Statusinformationen im *Store* aktualisiert werden müssen. Die Statusinformationen werden durch ein großes JavaScript-Objekt repräsentiert.

```
{
  counter: {
    count: 0
  },
  log: {
    infos: []
  }
}
```

Zum Code →

State-Object – einfaches Beispiel

Der Store soll zu jedem Zeitpunkt einen validen Zustand haben. Aus diesem Grund wird ein Store immer mit Initialdaten erstellt, um zu gewährleisten, dass bereits zum Start der Anwendung ein valider Anwendungszustand herrscht.

Damit kann eine *Component* jederzeit Daten aus dem Store abrufen.

```
this.store.subscribe(state => {
  console.log('Counter', state.counter.count);
  console.log('Log Messages', state.log.infos);
});
```

Zum Code →

Daten aus dem Store lesen

Wie bei der ***Action*** muss das ***State***-Objekt serialisierbar sein. Darüber hinaus gelten drei Prinzipien für den Redux Store.

Der Store

1. … ist Read-Only.
2. … ist die Single Source of Truth.
3. … wird nur über Pure Functions verändert.

Store Prinzipien

Read-Only

Informationen können im Store nicht einfach überschrieben werden. Dies geht nur über das Versenden einer ***Action.*** Das sichert den Store gegen versehentliche Mutationen ab. Das führt ebenfalls dazu, dass die Change-Detection in der gesamten Anwendung auf OnPush gestellt werden kann. Dadurch sinkt die Anzahl der Change-Detection-Cycles und führt zu einer besseren Performance in der Anwendung.

Single Source of Truth

Der Store soll in der Lage sein, jedwede Anfrage einer Component zu beantworten. Dazu müssen alle erforderlichen Informationen im Store hinterlegt werden.

Wenn Sie Redux konsequent einsetzen, werden Komponenten nur noch mit dem ***Store***-Service kommunizieren, um mit dem Anwendungsstatus zu arbeiten.

Pure Functions

Eine Statusänderung soll nur über Pure-Functions erfolgen. Diese Funktionen sind seiteneffektfrei. Das bedeutet, dass sie bei der gleichen Eingabe immer mit dem gleichen Resultat antworten, da sie nicht

von externen Quellen abhängig sind, die potenziell fehlschlagen oder unvorhersehbare Antworten liefern (Bsp.: HTTP-Client).

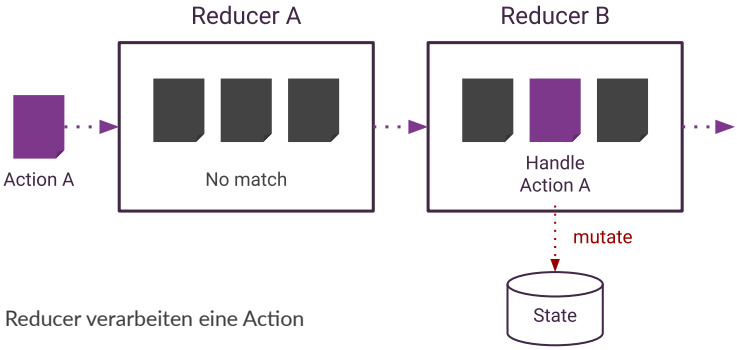
In der Abbildung ***Undirektionaler Redux-Datenfluss*** wurde der Begriff ***Reducer*** eingeführt. Die Reducer sind die angesprochenen Pure-Functions in Redux.

Aus den Prinzipien von Redux ergeben sich zwei Fragen
1. Wie fügen sich Pure-Functions in die Redux-Architektur ein?
2. Wie werden Seiteneffekte in Redux behandelt?

Beide Fragen werden in den nächsten zwei Abschnitten beantwortet.

Reducer

Eine ***Reducer***-Funktion hat die Aufgabe, versendete ***Actions*** zu verarbeiten. In einer Redux-Anwendung gibt es beliebig viele Reducer-Funktionen. Wird eine ***Action*** versendet, werden alle ***Reducer*** ausgeführt. Der jeweilige Reducer schaut zunächst auf die ***type***-Property der ***Action.*** Wenn es für den type eine zugehörige Mutation gibt, wird sie ausgeführt und der Status im Store aktualisiert.



Reducer verarbeiten eine Action

Die Abbildung *Reducer verarbeiten eine Action* zeigt, wie eine *Action A* versendet wird und mehrere Reducer passiert. Der *Reducer A* hat nichts mit der *Action* zu tun. Darum passiert in diesem *Reducer* nichts. In *Reducer B* gibt es dann ein „Match" und die *Action* resultiert in einer Zustandsveränderung.

Eine Redux-Anwendung verfügt über zahlreiche Reducer, weil jede dieser Funktionen für einen bestimmten Teil zuständig ist. Gibt es in Ihrer Anwendung einen Login, gibt es wahrscheinlich einen *AuthenticationReducer*. Verwalten Sie zudem Produkte oder Bestellungen, werden diese Bereiche durch einen *ProductsReducer* beziehungsweise einen *OrdersReducer* abgedeckt.

Gemäß dem ersten Prinzip von Redux ist der State *Read-Only*. Daher wird bei der Aktualisierung des Zustands der bestehende Zustand in eine neue Objektreferenz verpackt und die Mutation wird auf der neu entstandenen Kopie appliziert. Ab diesem Zeitpunkt repräsentiert die State-Kopie den neuen Anwendungszustand.

```
1  function counterReducer(state: State, action: Action): State {
2   switch (action.type) {
3     case '[Counter] Add':
4         return {
5           ...state,
6           count: state.count + action.payload
7         };
8     default: return state;
9   }
10 }
```

Zum Code →

Reducer-Funktion

Das Snippet ***Reducer-Funktion*** zeigt ein einfaches Beispiel einer Redux-Reducer-Funktion. Sie nimmt stets den aktuellen Anwendungszustand (hier ***state***) und die ***Action*** entgegen und gibt einen neuen ***State*** zurück. Falls der ***type*** der Action nicht passt, wird der ***State*** unverändert zurückgegeben.

Darüber hinaus hat eine ***Reducer***-Funktion noch eine weitere Aufgabe, die im vorangegangenen Snippet noch nicht verzeichnet ist. In jedem Reducer werden die jeweiligen Initialdaten festgelegt, damit der Store beim Start der Anwendung direkt einen validen Zustand hat.

```
1  const initialState: State = { count: 0 };
2
3  function counterReducer(
4    state = initialState,
5    action: Action
6  ): State {
7    // switch-case action.type ...
8  }
```

Zum Code →

Reducer-Funktion Initial State

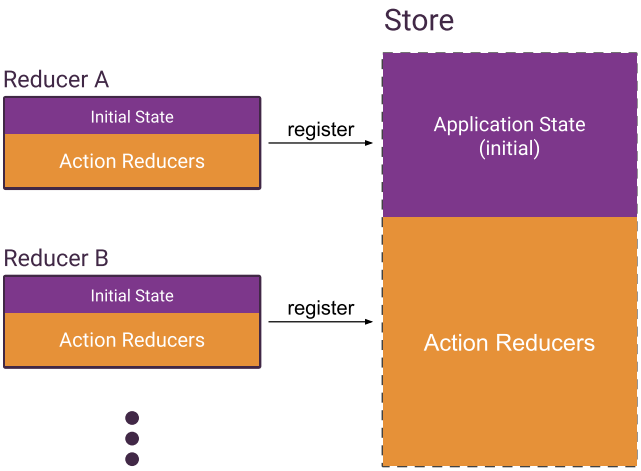
Jeder Reducer wird im ***Store*** registriert, damit eine versendete ***Action*** verarbeitet werden kann. In Angular erfolgt dies über die Konfiguration eines Moduls. Das nachstehende Snippet skizziert, wie die Registrierung aussieht.

```
1  StoreModule.forRoot({
2    counter: counterReducer,
3    log: logReducer
4  })
```

Zum Code →

Reducer-Funktionen registrieren

Die implementierten *Reducer* kommen im *Store* zusammen. Da jede *Reducer*-Funktion einen initialen *State* hat, entsteht aus der Summe aller *Reducer* der sogenannte *AppState*.

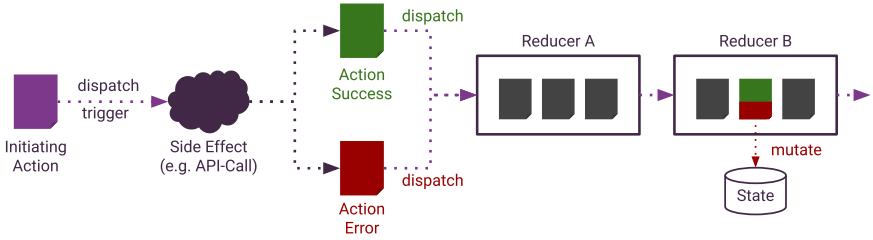


Die Gesamtheit der registrierten Reducer formen den Store

Die *Reducer*-Funktionen verarbeiten *Actions*, die im *Store* versendet werden. Diese drei Bausteine machen den synchronen, unidirektionalen Flow von Redux aus. Es fehlt nur noch eine letzte Komponente: das Behandeln von Seiteneffekten oder auch asynchronen Operationen.

Async-Flow – Effects

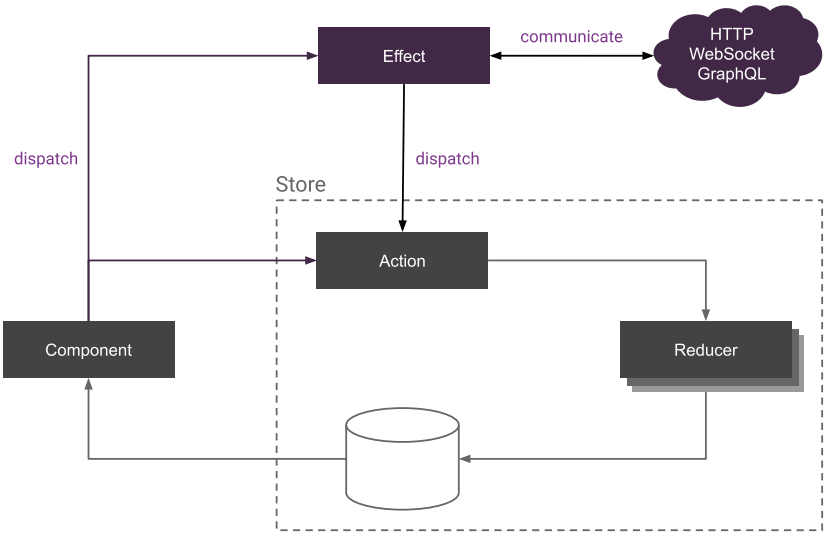
Der Redux-Zyklus soll seiteneffektfrei sein. Das heißt, dass diese Seiteneffekte isoliert werden müssen, damit Redux nicht an seiner naturgemäßen Stabilität einbüßt. Der Trick ist, dass diese „unsicheren“ Operationen mithilfe mehrerer *Actions* modelliert werden.



Asynchrone Operationen werden mit mehreren Actions modelliert

Es wird eine *Action* genutzt, um eine Operation zu initiieren. Häufig handelt es sich dabei um API-Aufrufe. Die Operation kann erfolgreich sein oder fehlschlagen. Für beide Szenarien können *Actions* bereitgestellt werden, die je nach Ausgang zum *Store* versendet werden, wo sie ein *Reducer* verarbeitet.

Das bedeutet, dass *Actions* nicht zwangsweise direkt zu einem *Reducer* gesendet werden. Sie können ebenso durch einen *Service* abgefangen werden. Das Resultat der initiierten Operation wird wieder in eine *Action* verpackt und zum *Store* gesendet.



Behandlung von Seiteneffekten mit Redux

Um einen ***Effect*** zu implementieren, muss es eine Möglichkeit geben, auf den Strom der ***Actions*** zu lauschen. Dann muss die initiierende ***Action*** herausgefiltert werden. Nachdem der Seiteneffekt fertig ausgeführt ist, wird je nach Ergebnis eine ***Action*** zum ***Store*** versendet.

Das folgende Snippet dient lediglich der Veranschaulichung. Es ist für den Gebrauch in produktivem Code ungeeignet. Die richtige Nutzung von Seiteneffekten wird im Abschnitt @ngrx/effects gezeigt.

```
storeActions.pipe(
  filter(action => action.type === 'Initiate'),
  switchMap(action => apiCall.execute(action.payload)),
  map(result => store.dispatch({
    type: 'Success', payload: result
  })),
  catchError(err => store.dispatch({
    type: 'ERROR', payload: err.message
  }))
)
```

Zum Code →

Stream zur Verarbeitung eines Seiteneffekts

Die NgRx Plattform

Die in diesem Abschnitt gezeigten Code-Beispiele gehören zu einer Beispielanwendung. Auf https://stackblitz.com/edit/ngrx-9-playground können Sie sich das NgRx Projekt ansehen.

NgRx Logo

NgRx ist ein Framework für Angular, das die Redux-Architektur implementiert. NgRx versteht sich jedoch nicht nur als Redux-Implementierung, sondern stellt reaktive Erweiterungen für Angular zur Verfügung.

Das Core-Team von NgRx stellt die Bibliothek als Plattform auf. Das bedeutet, dass sie die Grundlage schaffen, die anderen Teams gestattet, eigene Erweiterungen und Abstraktionen zu entwickeln. Schon längst wächst das Ökosystem um Frameworks, die NgRx nutzen. Dazu zählen NgRx Auto Entity und NgRx Ducks.

Die folgenden Abschnitte fokussieren sich jedoch auf die built-in Mechanismen von NgRx. Es wird gezeigt, wie die vorab besprochene Redux-Architektur durch NgRx umgesetzt wird.

Installation

NgRx ist in mehrere Pakete aufgeteilt. Über *npm* können alle Module installiert werden.

```
npm install @ngrx/{store,effects,entity,store-devtools,schematics}
```

Installation der gesamten NgRx-Suite

Modul	Funktion
@ngrx/store	Stellt Actions, Reducer, Selectors und Store bereit. Mit diesen Werkzeugen kann der synchrone Flow von Redux umgesetzt werden.
@ngrx/effects	Bietet eine API für das Handling von Seiteneffekten.
@ngrx/entity	Vereinfacht das Schreiben von Mutationen in Reducern und stellt fertige Selektoren bereit.
@ngrx/store-devtools	Erlaubt komfortables Debugging mithilfe einer Browser-Extension.
@ngrx/schematics	Erweiterung für Angular CLI, um Code erzeugen zu lassen.
@ngrx/router-store	Synchronisiert Location-Status mit dem Store. Nicht Teil dieses Artikels (siehe https://ngrx.io/guide/router-store für mehr Informationen).
@ngrx/data	Automatisiert Erzeugung von Effekten und Reducern für CRUD-Szenarien. Nicht Teil dieses Artikels (siehe https://ngrx.io/guide/data für mehr Informationen).

Die NgRx-Module im Überblick

Wie bei jeder höheren Architektur ist es notwendig, die erforderliche Infrastruktur bereitzustellen. Bei dieser Aufgabe unterstützen die *Schematics* von NgRx. Die Kommandos erzeugen den Code, den es braucht, um NgRx in der Angular-App zu nutzen. Darüber hinaus bietet es Code-Templates an, die die Implementierung neuer Features beschleunigen.

@ngrx/store

Um den NgRx-Store zu initialisieren, wird folgendes Kommando der *@ngrx/schematics* ausgeführt.

```
ng generate store State
```

Initialisierung des NgRx Stores

Neben der Erzeugung einiger Verzeichnisse werden in der *app.module.ts* einige NgRx-Module eingetragen. Das dient dazu, die Dienste in Angular zu registrieren und den Store zu konfigurieren.

```
1  StoreModule.forRoot(reducers, {
2     metaReducers,
3     runtimeChecks: {
4       strictStateImmutability: true,
5       strictActionImmutability: true,
6     }
7   })
```

Zum Code →

Setup des NgRx Stores im App Module

Im *StoreModule* werden *reducers* registriert. Darüber hinaus können in einem Konfigurationsobjekt *metaReducers* genutzt werden. Diese kann man sich wie Plugins für den Store vorstellen. Es sind ebenfalls *Reducer-Functions*. Ein gängiger Anwendungsfall ist ein Logger, der jede *Action* protokolliert.

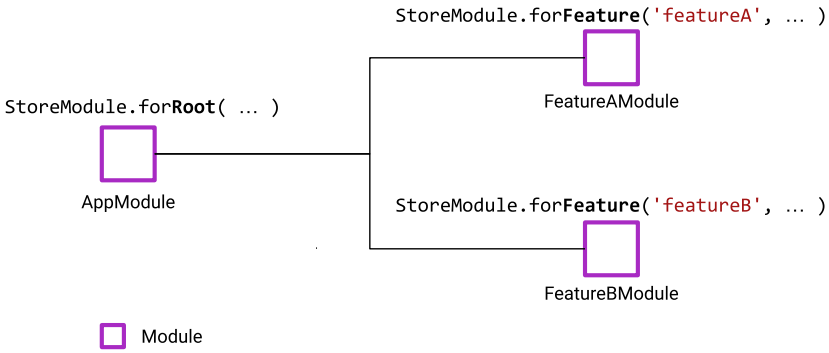
Im *Root-StoreModule* werden *runtimeChecks* konfiguriert. Die Optionen *strictState-* und *strictActionImmutability* sorgen dafür, dass das erste

Prinzip von Redux eingehalten wird. Sollte dagegen verstoßen werden, kommt es zu einem Fehler, der auf der Konsole zu sehen ist.

Es fällt auf, dass das ***StoreModule*** nicht direkt genutzt wird, sondern dass die statische Methode ***forRoot*** aufgerufen wird, um den ***Store*** zu initialisieren.

NgRx folgt dem Muster des Angular-Routers (vgl. @angular/router – ForRoot-Pattern). Es gibt ein Root-Modul, das die Services lädt, die für den Betrieb des Stores gebraucht werden. Zusätzlich können sog. ***Features*** hinzugefügt werden, die in den jeweiligen ***Feature-Modules*** registriert werden können.

NgRx unterstützt Lazy-Loading. Wenn ein Feature-Modul erst nachträglich geladen wird, wird das Feature zur Laufzeit dem Store hinzugefügt.



Root- & Feature-Stores

Jedes *Feature* erhält einen eineindeutigen Namen (hier: *featureA*, *featureB*). Unter diesem Namen werden die Daten im Store verwaltet.

```
{
  featureA: { ... },
  featureB: { ... }
}
```

Zum Code →

State-Object – Verwendung von Features

Was gehört in den Root-Store und was in die Feature-Stores?
Der *Root-Store* stellt Daten bereit, die von allen *Features* geteilt werden. Dazu gehören Location-State oder auch der Session-State. Diese Informationen werden auch von anderen Modulen benötigt. Daher ist es gut, wenn diese Informationen gleich zu Beginn zur Verfügung stehen.

In einem *Feature-Store* wird der State des jeweiligen Features verwaltet.

Beim Hinzufügen eines neuen Features unterstützen die Schematics ebenfalls.

```
ng generate module counter
ng generate @ngrx/schematics:feature counter/store/counter \
--module counter/counter.module.ts
```

Initialisierung eines NgRx Features

Falls noch nicht vorhanden wird ein Modul angelegt. Anschließend wird das *Feature* diesem Modul zugeordnet. Es werden alle benötigten Dateien generiert. Außerdem werden Templates für Unit- und Integrationstests zur Verfügung gestellt.

```
counter/
├ store/
│ ├ counter.actions.spec.ts
│ ├ counter.actions.ts
│ ├ counter.effects.spec.ts
│ ├ counter.effects.ts
│ ├ counter.reducer.spec.ts
│ ├ counter.reducer.ts
│ ├ counter.selectors.spec.ts
│ └ counter.selectors.ts
├ counter.component.ts
└ counter.module.ts
```

Zum Code →

NgRx-Building-Blocks in einem Feature Module

In der Datei *counter.module.ts* werden die Reducer ebenfalls unter dem *Feature-Key* (hier: *counter*) registriert.

```
import { StoreModule } from '@ngrx/store';
import * as fromCounter from './store/counter.reducer';

// … @NgModule Definition

StoreModule.forFeature(
  fromCounter.counterFeatureKey,
  fromCounter.reducer
)
```

Zum Code →

counter.module.ts

Es gibt eine eigene Datei für *Actions*, die nun exemplarisch um eine Inkrement- und Decrement-*Action* erweitert wird.

Anstatt das Action-Objekt manuell zu erzeugen, bietet NgRx Hilfsmethoden an, die die Deklaration und Typisierung vereinfachen.

```
1  import { createAction, props } from '@ngrx/store';
2
3  export const increment = createAction(
4   '[Counter] Increment',
5   props<{ payload: number }>()
6  );
7
8  export const decrement = createAction(
9   '[Counter] Decrement',
10  props<{ payload: number }>()
11 );
```

Zum Code →

counter.actions.ts

Der Helfer *createAction* gibt eine Funktion zurück, mit der die jeweilige *Action* erzeugt werden kann. Darüber hinaus wird die Funktion *props<T>* eingesetzt, um den Typen der *payload* festzulegen.

Die *Actions* können mithilfe des *Store*-Services von NgRx versendet werden. Der *Store* selbst ist ein Observable. Das heißt, dass Daten als Stream in der Komponente gebunden und visualisiert werden können.

Im folgenden Snippet wird der Type any temporär verwendet, um die Erläuterungen etwas abzukürzen. Diese Situation wird durch den Einsatz von *NgRx-Selectors* noch behoben.

```ts
import { Component } from '@angular/core';
import { Store, select } from '@ngrx/store';
import { Observable } from 'rxjs';
import { decrement, increment } from './store/counter.actions';

@Component({
 selector: 'app-counter',
 template: `
   <strong>{{ count | async }}</strong>
   <button (click)="increment(1)">Increment</button>
   <button (click)="decrement(1)">Decrement</button>
 `
})
export class CounterComponent {
 count: Observable<number>;

 constructor(private store: Store) {
   this.count = this.store.pipe(
     select((state: any) => state.counter.count)
   );
 }

 increment(payload: number) {
   this.store.dispatch(increment({ payload }));
 }
 decrement(payload: number) {
   this.store.dispatch(decrement({ payload }));
 }
}
```

Zum Code →

counter.component.ts

Im Konstruktor der *CounterComponent* wird der aktuelle Zählerstand abonniert. Sofern er sich ändert, wird die Ansicht aktualisiert. Die Änderung von *count* wird über die *Actions increment* und *decrement* hervorgerufen, die in den jeweiligen Methoden durch einen Button-Klick versendet werden.

Damit sich *count* tatsächlich erhöht oder senkt, müssen die *Actions* in einer *Reducer*-Funktion verarbeitet werden.

Im Theorieteil wurde ein *Reducer* mit einem *switch-case*-Statement implementiert. NgRx bietet eine alternative API an, die besser typisiert ist und deren Ziel es ist, die Lesbarkeit des Codes zu erhöhen.

```
1
2  import { createReducer, on } from '@ngrx/store';
3  import { increment, decrement } from './counter.actions';
4
5  export const initialState: State = {
6   count: 0
7  };
8
9  export const reducer = createReducer(
10  initialState,
11
12  on(increment, (state, { payload }) => ({
13    ...state,
14    count: state.count + payload
15  })),
16
17  on(decrement, (state, { payload }) => ({
18    ...state,
19    count: state.count - payload
20  }))
21 );
```

Zum Code →

counter.reducer.ts

Mithilfe von ***createReducer*** wird eine ***Reducer***-Funktion erstellt. Der erste Parameter repräsentiert den initialen Zustand beim Start der Anwendung. In diesem Beispiel wird er dazu genutzt, um den ***count*** auf 0 zu setzen. Danach können beliebig viele ***Actions*** behandelt werden. Dafür wird on verwendet. Die Methode nimmt die zu behandelnde ***Action*** entgegen. Danach wird die Methode implementiert, die den ***State*** verändert (hier: Addition und Subtraktion).

Der erzeugte ***Reducer*** wird in der ***counter.module.ts*** durch ***StoreModule.forFeature***(...) im Store registriert.

Im Reducer wird der Spread-Operator (...state) genutzt, um eine Kopie des Anwendungszustands zu erzeugen. Die Mutation wird dann auf der Kopie angewandt und ist ab diesem Zeitpunkt der neue State. Hier wird dem ersten Prinzip von Redux Rechnung getragen. Der ***State*** ist Read-Only.

Mit den bisher gezeigten Code-Beispielen funktioniert das Counter-Feature bereits. Um Daten vom Store abzurufen, stellt NgRx ein weiteres Werkzeug zur Verfügung: ***Selectors.***

Ein ***Selector*** ist eine Projektionsfunktion, um bestimmte Daten aus dem Store zu lesen. Der Einsatz von ***Selectors*** hat zwei Vorteile:

1. Die ***Component*** nutzt die Projektion des ***Selectors*** und hat keine Kenntnis mehr, wie der Store intern strukturiert ist. Das erlaubt Optimierungen am Store, ohne dass die ***Component*** angepasst werden muss.
2. ***Selectors*** können miteinander kombiniert werden und erlauben das Aggregieren von Daten aus unterschiedlichen Teilen des ***Stores.***

Um einen *Selector* zu erzeugen, wird zunächst ein *Feature-Selector* gebraucht. Dieser ist wie ein Lesezeichen, das zu einem bestimmten Teil des Stores springt, um von diesem Punkt aus Daten zu selektieren. Dafür stellt NgRx die Methode *createFeatureSelector<T>* zur Verfügung. Diese Methode erwartet den *Feature-Key*. Der *Feature-Selector* muss an dieser Stelle manuell typisiert werden, damit die Auto-Vervollständigung weiterhin funktioniert und Laufzeitfehler vermieden werden.

Der von der *Component* genutzte *Selector* wird mit *createSelector* erzeugt. Diese Funktion erwartet den *Feature-Selector* und erlaubt anschließend das Definieren einer Projektionsfunktion, die genau die Daten liest, die von der *Component* benötigt werden.

```
1  import { createFeatureSelector, createSelector } from '@ngrx/store';
2  import * as fromCounter from './counter.reducer';
3
4  export const counterState = createFeatureSelector<fromCounter.State> (
5    fromCounter.counterFeatureKey
6  );
7
8  export const count = createSelector(
9    counterState,
10   state => state.count
11 );
```

Zum Code →

counter.selectors.ts

Durch die Bereitstellung des *Selectors count*, kann der Code in der *counter.component.ts* vereinfacht werden.

```
constructor(private store: Store) {
  // Vorher
  this.count = this.store.pipe(
     select((state: any) => state.counter.count)
   );
  // Nachher
  this.count = this.store.pipe(select(count));
}
```

Zum Code →

counter.component.ts

Das Code-Beispiel zeigt, dass der *select*-Operator von *@ngrx/store* sowohl Inline-Projektionen als auch *Selector*-Funktionen akzeptiert.

Die in diesem Abschnitt gezeigten Instrumente von NgRx bilden den synchronen Flow von Redux ab. Der folgende Abschnitt zeigt, wie Seiteneffekte behandelt werden.

@ngrx/effects

Für die Behandlung von Seiteneffekten stellt NgRx ein eigenes Modul bereit: @ngrx/effects. Damit *Effects* in der Angular-Anwendung funktionieren, muss das *EffectsModule.forRoot([])* einmalig in der App registriert werden. Wie auch beim *StoreModule* werden dadurch alle erforderlichen Services bereitgestellt.

```
EffectsModule.forRoot([])
```

Zum Code →

app.module.ts

Anschließend kann jedes Feature seine eigenen Effects bereitstellen. Diese sind nichts anderes als Services, die auf den Strom von *Actions* lauschen und bei bestimmten *Actions* (den Initiating Actions) asynchrone Operationen ausführen. Jeder Effect-Service muss im *EffectsModule* registriert werden.

```
1   EffectsModule.forFeature([CounterEffects])
```

Zum Code →

counter.module.ts

Die @ngrx/effects bieten einen injizierbaren Service an, der den *Action*-Stream bereitstellt. In Verbindung mit der Funktion *createEffect* wird ein *Effect* bereitgestellt. Der Stream von *Actions* wird mithilfe des Operators *ofType* gefiltert. So wird die Pipe des Streams nur dann weiterverarbeitet, wenn die gewünschte Action versendet wurde (hier: *randomAdd*).

Im folgenden Beispiel wird eine *Initiating Action* mit der Bezeichnung *randomAdd* verarbeitet. Es wird eine zufällig generierte Nummer erzeugt. Die Funktion *randomizedNumber* kann allerdings auch einen Fehler hervorrufen. Das folgende Snippet zeigt, dass bei erfolgreicher Generierung der Nummer die *Action add* erzeugt wird. Tritt ein Fehler auf, wird eine Action *logInfo* zurückgegeben, die den Fehler protokolliert.

Im *Effect* selbst ist jedoch kein *this.store.dispatch()* zu sehen. Das liegt daran, dass die Werte, die durch den *Effect*-Stream zurückgegeben werden, durch NgRx selbst zum Store verschickt werden. Das bedeutet weniger Aufwand beim Programmieren für den Entwickler.

```
@Injectable()
export class CounterEffects {
 randomAdd = createEffect(() =>
   this.actions.pipe(
     ofType(randomAdd),
     concatMap(() =>
       randomizedNumber().pipe(
         map(value => add({ payload: { value } })),
         catchError(message => of(logInfo(message)))
       )
     )
   )
 );

 constructor(private actions: Actions) {}
}
```

Zum Code →

counter.effects.ts

In manchen Fällen ist es nicht gewünscht, dass ein *Effect* eine *Action* automatisch versendet. Die Funktion *createEffect* akzeptiert einen zweiten Parameter, der zur Konfiguration dient. Damit kann das Versenden deaktiviert werden.

```
randomAdd = createEffect(() =>
  this.actions.pipe(...),
  { dispatch: false }
);
```

Zum Code →

Effect – Das Versenden einer Action deaktivieren

Der gezeigte Code ist Teil einer Beispielanwendung, die Sie auf https://stackblitz.com/edit/ngrx-9-playground finden.

Der Umgang mit ***Effects*** ist die größte Herausforderung in NgRx. ***Services*** können lose miteinander gekoppelt werden und je nach Ergebnis der Operation können die passenden ***Actions*** versendet werden, um das Verhalten der Anwendung steuern.

@ngrx/entity

In diesem Artikel wurde bereits viel von der Verwaltung des Anwendungsstatus gesprochen. Vielleicht ist diese Ausdrucksweise etwas zu hoch gegriffen. Im Grunde geht es um das Hinzufügen, Bearbeiten, Löschen und Lesen von Datensätzen. Häufig müssen diese Operationen an Listen durchgeführt werden. Der dafür benötigte Code ist oft sehr ähnlich.

Darum hat das NgRx-Core Team mit @ngrx/entity eine Bibliothek geschaffen, die wiederkehrende Arbeiten in ***Reducer***-Funktionen und ***Selectors*** vereinheitlichen.

Listen können mit @ngrx/entity effektiv verwaltet werden. Dafür wird das Interface ***EntityState<T>*** zur Verfügung gestellt. Dessen Einsatz homogenisiert den Aufbau des States. Anstelle die Elemente in einem Array abzuspeichern, wird ein Objekt mit einer Key-Value-Zuweisung genutzt. Der ***Key*** ist die Id des jeweiligen Datensatzes. Das bedeutet, dass die Elemente (Entities genannt) über ein Feld mit einem eindeutigen ***Identifier*** verfügen müssen.

Nun können alle mutierenden Operationen auf die gleiche Art und Weise ausgeführt werden. Der ***EntityAdapter<T>*** liefert alle herkömmlichen Funktionen ***(Create, Update und Delete)***, um den ***State*** zu aktualisieren.

```typescript
import { EntityState } from '@ngrx/entity';

export interface Message {
   id: string;
   text: string;
 }

export interface LoggerState extends EntityState<Message> {}
```

Zum Code →

State mithilfe von EntityState<T> definieren

Ein *EntityAdapter* wird mit der Funktion *createEntityAdapter<T>* erstellt und typisiert. Drei wesentliche Funktionen stehen dann zur Verfügung:

1. Mutationsfunktionen können im *Reducer* genutzt werden.
2. *InitialState* kann für den *Reducer* generiert werden.
3. *Selectors* werden zur Verfügung gestellt.

```typescript
import { createEntityAdapter, ... } from '@ngrx/store';

export const adapter = createEntityAdapter<Message>();

export const loggerReducer = createReducer(
 adapter.getInitialState(),
 on(logInfo, (state, { payload }) => adapter.addOne(payload, state))
);
```

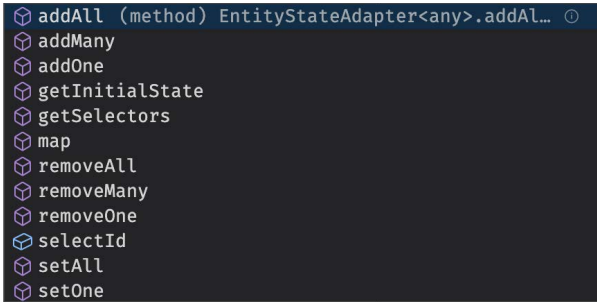
Zum Code →

Einsatz des EntityAdapters

Das Codebeispiel zeigt, dass der *EntityAdapter* unter anderem eine Funktion *addOne* bereitstellt, mit dem ein Element einer Liste hinzugefügt wird. Neben der Operation sorgt der *EntityAdapter* auch dafür, dass eine Kopie des *States* angelegt wird. Der Entwickler muss damit nicht mehr an den Spread-Operator denken.

Der *EntityAdapter* hat neben *addOne* noch weitere Funktionen zu bieten, die das Entwickeln von *Reducern* einfacher machen.

```
export const loggerReducer = createReducer(
  adapter.getInitialState(),
  on(logInfo, (state, { payload }) => adapter.addOne(payload, state))
);
```



EntityAdapter – Die Funktionen im Überblick
(Details unter: https://ngrx.io/guide/entity/adapter#adapter-collection-methods)

Da durch das EntityState-Interface geregelt ist, wie die Listenelemente gespeichert werden, ist die Selektion auch immer gleich. Der *EntityAdapter* liefert fertige *Selectors* aus, die einzig und allein mit dem *FeatureSelector* verknüpft werden müssen, damit die Informationen vom richtigen Ort aus dem Store selektiert werden können.

```
import { adapter } from './logger.reducer';

export const loggerFeatureKey = 'logger';

const featureLogger = createFeatureSelector<LoggerState>(loggerFeatureKey);

export const {
 selectAll,
 selectTotal,
 selectEntities,
 selectIds
} = adapter.getSelectors(featureLogger);
```

Zum Code →

EntityAdapter – Vorbereitete Selektoren bereit zum Einsatz

Mit @ngrx/entity werden kleine Helfer zur Verfügung gestellt, die nicht nur die Produktivität steigern, sondern auch den Code lesbarer machen.

Bei all den neuen Techniken, die bisher diskutiert wurden, bringt Ihnen der Einsatz von Redux auch an der Tooling-Front Vorteile. Im folgenden Abschnitt lernen Sie die ***StoreDevtools*** von NgRx kennen.

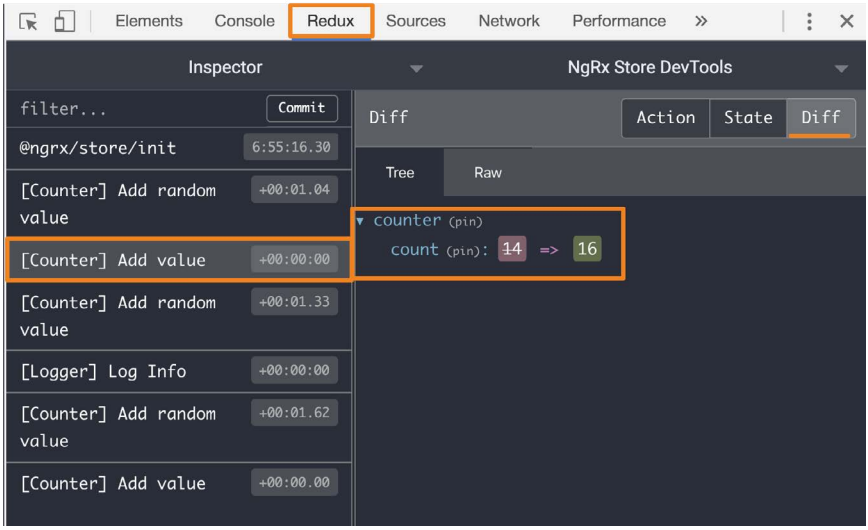
@ngrx/store-devtools

Für Anwendungen, die Redux einsetzen, gibt es Erweiterungen für Firefox und Chrome, die ***Actions*** und Statusänderungen visualisieren.

So haben Sie stets einen guten Überblick über das Geschehen in Ihrer App. Es ist leicht zu prüfen, ob die ***Reducer*** arbeiten, wie erwartet und ob die richtigen ***Actions*** versendet wurden.

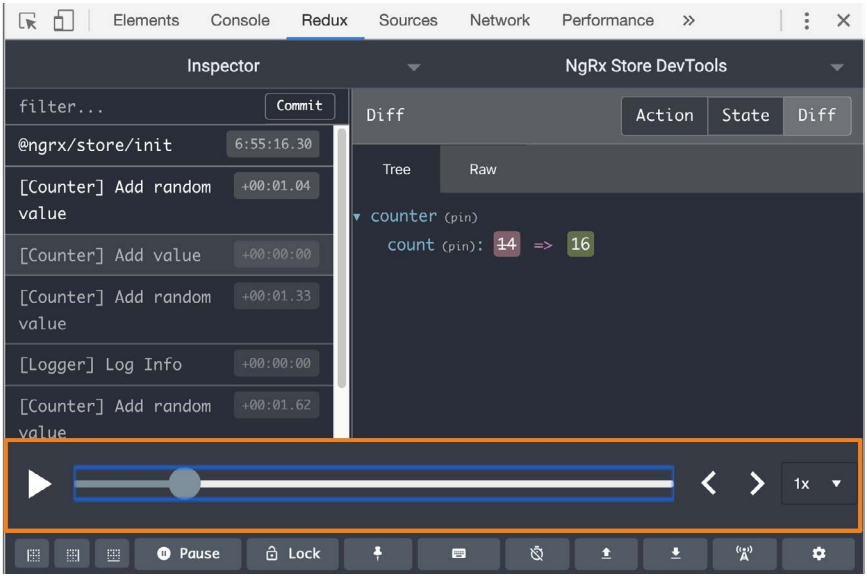
Nach der Installation taucht in den Developer Tools ein neuer Reiter mit dem Titel *Redux* auf. Mit einem Klick wird eine dunkle Oberfläche sichtbar, die auf der linken Seite alle versendeten *Actions* als Log präsentiert.

Wird eine *Action* ausgewählt, wird auf der rechten Seite ein *Diff* angezeigt. Es zeigt die Änderung an, die durch die *Action* im Store hervorgerufen wurde.



Redux DevTools – Übersicht

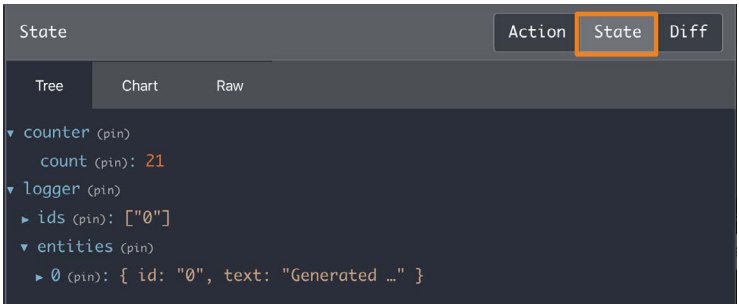
Der untere Bereich der DevTools sieht etwas wie ein Musikplayer aus. Dieses Control erlaubt Time-Travel-Debugging. Sie können versendete *Actions* und die damit einhergegangenen Änderungen zurückspulen. Die zurückgenommen *Actions* werden in den DevTools ausgegraut.



Redux DevTools – Time-Travel-Debugging

Sobald Ihr NgRx-Projekt und die Anzahl der *Actions* wächst, können Sie in Ruhe alle Prozesse analysieren und nachvollziehen.

Neben der Diff-Anzeige ist es auch jederzeit möglich, den gesamten *State* anzeigen zu lassen. Dazu kann in der oberen, rechten Ecke die Ansicht gewechselt werden.



Redux DevTools – Anzeige des App-States

Die Redux-DevTools bieten auch noch andere Funktionen, wie das Importieren oder Exportieren der *Actions*. Alle Funktionen sind nur einen Klick entfernt und warten darauf, ausprobiert zu werden.

Die Redux-DevTools wurden unabhängig von NgRx entwickelt. NgRx nutzt die API der DevTools, um diese zu nutzen. Dafür muss *StoreDevtoolsModule.instrument()* einmalig in einem Modul konfiguriert werden.

Wenn Sie die @ngrx/schematics einsetzen, werden die DevTools automatisch bereitgestellt. Die Standardkonfiguration sieht vor, dass die DevTools nur im Development-Mode von Angular zur Verfügung gestellt werden.

```
1  !environment.production
2     ? StoreDevtoolsModule.instrument()
3     : []
```

Zum Code →

Redux DevTools – Registrierung in app.module.ts

Fazit
Redux im Allgemeinen und NgRx im Speziellen haben zu Beginn eine steile Lernkurve. Der Code ist fragmentierter als zuvor, und es ist nur verständlich, dass es kurzzeitig frustrierend sein könnte.

Mit zentralem StateManagement ist das Entwickler-Team im Leistungssport angekommen und die jeweiligen Disziplinen müssen anfangs trainiert werden. Sobald die Muscle-Memory da ist, können Änderungen und Neuerungen schnell umgesetzt werden. Das liegt nicht zuletzt daran, dass Redux das CQS-Pattern (Command-Query-Segregation) forciert.

Das NgRx-Team hat viel Arbeit in die Typisierung gesteckt, sodass Entwickler schnell darauf aufmerksam werden, wenn Bestandteile falsch miteinander verdrahtet sind. Außerdem helfen Tools wie die @ngrx/schematics oder die @ngrx/store-devtools dabei, Code schnell zu generieren und zur Laufzeit zu analysieren.

Die Tatsache, dass Seiteneffekte isoliert werden, wird zu Anfang häufig etwas unterschätzt. Sie werden schnell feststellen, welch ein Segen es ist, dass Services austauschbarer werden, beziehungsweise schnell neu miteinander verdrahtet werden können, wenn es die Anforderungen vorgeben. Die @ngrx/effects wirken hier wie ein zustandsloser Composition-Layer.

Wenn Sie vor der Entscheidung stehen, ob Sie NgRx einsetzten wollen, bietet sich ein Review des Codes an. Wenn die folgenden Fragen größtenteils mit „Ja“ beantwortet wird, sollte überlegt werden den Schritt zum zentralen State-Management zu machen:

- Ist das Service-Composition-Anti-Pattern bereits stark verbreitet?
- Sind Methoden in Komponenten und Services sehr komplex? (Messbar mit der zyklomatischen Komplexität)
- Sind Services untereinander gekoppelt?
- Existieren mehrere Services die bereits Statusinformationen verwalten?
- Ist die Aggregation verschiedener Datenquellen sehr komplex?
- Wird es zunehmend schwieriger Bestandteile in der Anwendung schnell auszutauschen?

NgRx macht Ihr Projekt definitiv komplexer, verschafft Ihnen und Ihrem Team jedoch eine hohe Flexibilität und stellt einen robusten Architekturunterbau für Ihre Anwendung zur Verfügung.

Referenzen

- Redux: https://redux.js.org
- NgRx: https://ngrx.io

Erweiterungen

- NgRx Auto Entity: https://briebug.gitbook.io/ngrx-auto-entity/
- NgRx Ducks: https://co-it.gitbook.io/ngrx-ducks

Ergänzende Artikel

- How to start flying with NgRx: https://indepth.dev/how-to-start-flying-with-angular-and-ngrx/
- Start using NgRx Effects: https://indepth.dev/start-using-ngrx-effects-for-this/
- NgRx 8's new factory methods: https://medium.com/@gregor.woiwode/ngrx-8-meet-the-new-upcoming-factory-methods-of-the-next-major-release-a97a079cc089
- Managing different Slices of State: https://indepth.dev/managing-different-slices-of-the-same-ngrx-state/
- NgRx creator functions: https://indepth.dev/ngrx-creator-functions-101/
- How and where handling loading and error states: https://indepth.dev/ngrx-how-and-where-to-handle-loading-and-error-states-of-ajax-calls/

Videos

- Good Action hygiene: https://youtu.be/JmnsEvoy-gY
- Selectors: https://youtu.be/Y4McLi9scfc
- Reduce Boilerplate: https://youtu.be/t3jx0EC-Y3c
- State Management Patterns & Best Practises: https://youtu.be/FQ6fzkHvCEY

Die Autorin

Martina Kraus

Schon seit frühen Jahren beschäftigt sich Martina Kraus mit der Webentwicklung. Das Umsetzen großer Softwarelösungen in Node.js und Angular hat sie schon immer begeistert. Neben ihrer Tätigkeit als private Dozentin im Bereich Webentwicklung an der Hochschule Mannheim arbeitet sie hauptberuflich als Softwareentwicklerin, vornehmlich mit Angular und C++. In ihrer Rolle als Google Developer Expert (GDE) liebt sie darüber hinaus, auf nationalen und internationalen Konferenzen das Wissen rund um Angular zu verbreiten, organisiert regelmäßig ngGirls Events (freie Angular Workshops für Frauen) und das lokale Angular Heidelberg Meetup.

Angular-Apps mit der Ivy Rendering-Engine

Möchte man heutzutage eine Single Page Application im Web-Browser entwickeln, sind Frameworks wie Angular kaum noch wegzudenken. Das Behandeln von User Interaktionen, wie sie beispielsweise bei einem Formular oder durch das Klicken auf ein Steuerelement vorkommen, werden durch Angular deutlich erleichtert.

Die Entwicklung mit Frameworks stellt Web-Entwickler vor ganz neue Herausforderungen: Wurden vor 10 Jahren noch viele einzelne HTML-Seiten an den Web-Browser ausgeliefert, wird bei einer Single Page Application nur noch eine einzige, sogenannte ***index.html*** ausgeliefert und sämtlicher Inhalt mithilfe von JavaScript neu gerendert.

Die Hauptverantwortlichkeit für das Darstellen der Webseite liegt nun im Web-Browser, und das dazu benötigte JavaScript wird vollständig ausgeliefert.

Leider bedeutet das auch, dass bei einer schwachen Internetverbindung die sogenannte ***initiale Ladedauer*** einer Anwendung bei einem sehr großen JavaScript-Bundle, welches heruntergeladen werden muss, sehr lang sein kann.

JavaScript wurde somit zur „kostbarsten" Ressource. In der Entwicklung von Webanwendungen möchte man natürlich die ausgelieferte Größe des JavaScript-Bundles so gering wie möglich halten.

Auch das Angular-Team nahm sich dieser Herausforderung gezielt an und entwickelte einen vollständig neuen Compiler: Ivy.

Während mit Angular 8 Ivy nur als Opt-in-Möglichkeit zur Verfügung stand, entweder durch das Setzen des *--enable-Ivy* Flags bei der Erstellung des Projektes, oder mit Hilfe der *compileroptions* in der *tsconfig.json*, wird nun der neue Angular-Compiler per Default ab Version 9 ausgeliefert.

Neben der verringerten Bundle-Größe bringt Ivy eine Reihe von weiteren Optimierungen mit sich:

- Schnellere Rebuilds durch die separate Kompilierung jeder einzelnen Datei ohne weitere Abhängigkeiten
- Einfache Gestaltung des Debugging durch Breakpoints im Template
- Verbesserte Typprüfung in den Templates

Dieser Artikel soll einige der Optimierungen des neuen Compilers näher beleuchten und Aufschluss über die neuen Möglichkeiten geben, die uns nun als Angular-Entwickler mit Ivy zur Verfügung stehen.

Aufgaben und Vorteile der neuen Komponentenentwicklung

Mit Ivy führt das Core-Team nun schon den dritten Angular-Compiler in das Framework ein (in Version 2 wurde die Template Engine als Compiler eingeführt und ab Version 4 die sogenannte ViewEngine, welche nun in Version 9 von Ivy vollständig abgelöst wird).

Doch was genau ist ein Compiler eigentlich, und welche Aufgaben verbergen sich hinter der Kompilierung des Angular-Codes?

In Angular schreiben wir einen sogenannten Template Code, welcher definiert und deklariert, wie unsere Komponente später im Web-Browser gerendert werden soll. Der Compiler generiert hieraus die gesamte DOM-Struktur und verknüpft diese mit den Daten, die wir programmatisch

mithilfe des Controller-Codes hineingeben. Hier sehen wir ein einfaches Beispiel eines Angular-Template-Codes und dessen JavaScript-Code, wie ihn ein Compiler beispielsweise generiert:

```
1  //Angular declarative Code
2  <div>{{ title }}</div>
3
4  //Instruction to create DOM
5  const div = document.createElement('div')
6  div.innerText = ctx.title
```

Zum Code →

Mithilfe dieses JavaScript Codes erhält unser Web-Browser nun genaue Anweisungen, wie er den HTML-Baum unserer Anwendung aufbauen soll.

Da nicht jeder Browser bereits die aktuellste ECMAScript-Version ausführen kann, muss man gegebenenfalls die benötigte Version, zu der unser Angular-Code kompiliert werden soll, in der ***tsconfig.json*** setzen (zu finden als sogenannte „target"-Version). Die Webseite CanIUse gibt hierbei eine gute Übersicht über die JavaScript-Features und welche Web-Browser diese bereits unterstützen.

Neben der Transformierung des Angular-spezifischen Codes in hocheffizienten und performanten JavaScript-Code bietet ein Compiler darüber hinaus noch einige weitere Funktionalitäten. So führt er beispielsweise auch eine statische Code-Analyse durch und stellt neben der Datentyp-Überprüfung sicher, dass ein Objekt auch tatsächlich die Attribute besitzt, welche man versucht, darauf aufzurufen.

Die genaue Vorgehensweise würde hierbei den Rahmen dieses Artikels sprengen. Für einen tieferen Einblick empfehle ich den Talk „Deep Dive into the Angular Compiler“ von Alex Rickabaugh auf der AngularConnect 2019.

Der generierte Code wird zusätzlich komprimiert, minifiziert und unleserlich gemacht und als das bereits erwähnte JavaScript-Bundle an den Web-Browser ausgeliefert.

Verkleinerung des JavaScript-Bundles

Einer der Hauptgründe für die Entwicklung eines neuen Angular-Compilers war die Verringerung der Größe eben dieses Bundles.

Um also zu verstehen, warum es mit Angular Ivy gelingt, ein kleineres JavaScript-Bundle zu generieren, müssen wir den generierten JavaScript-Code genauer unter die Lupe nehmen. Während man bei Angular 8 noch standardmäßig mit der ViewEngine als Angular-Compiler arbeiten musste und Ivy nur als Opt-In Möglichkeit zur Verfügung stand, bekommt man nun mit Angular 9 Ivy als Standard-Compiler mitgeliefert.

Wie man sein bestehendes Angular-Projekt auf die neue Version upgradet, kann in der offiziellen Angular-Dokumentation nachgelesen werden.

Für einen ersten einfachen Vergleich habe ich ein neues Projekt mit Angular Version 9 angelegt und es jeweils einmal mit Ivy und einmal mit der ViewEngine kompiliert.

Schauen wir uns die entsprechende *package.json* an:

```
  "dependencies": {
    "@angular/animations": "~9.0.2",
    "@angular/common": "~9.0.2",
    "@angular/compiler": "~9.0.2",
    "@angular/core": "~9.0.2",
    "@angular/forms": "~9.0.2",
    "@angular/platform-browser": "~9.0.2",
    "@angular/platform-browser-dynamic": "~9.0.2",
    "@angular/router": "~9.0.2",
    "rxjs": "~6.5.4",
    "tslib": "^1.10.0",
    "zone.js": "~0.10.2"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.900.3",
    "@angular/cli": "~9.0.3",
    "@angular/compiler-cli": "~9.0.2",
```

Zum Code →

Package.json unseres Beispielprojektes mit Angular 9

Um dieses Projekt mit Ivy zu kompilieren, musste ich nichts Weiteres tun, als den Angular-Compiler manuel innerhalb des Roots-Verzeichnisses meiner Angular-Applikation aufzurufen:

> ng new ivy-app
> cd ivy-app
> ngc

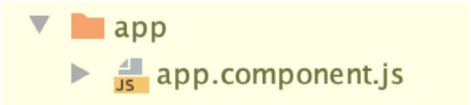
Um dasselbe Projekt nun mit der ViewEngine kompilieren zu können, musste ich zunächst Ivy in den *CompilerOptions* der *tsconfig.json* deaktivieren. Mit einem erneuten Aufruf des Compilers erhielt ich daraufhin den von der ViewEngine kompilierten JavaScript Code.

Bei einem direkten Vergleich der beiden Kompilate sieht man bereits, dass der neue Angular-Compiler hält, was er verspricht (das Kompilat liegt hierbei in einem nicht minifizierten oder komprimierten Format vor). So ist der mit Ivy generierte Code in unserem Beispielprojekt nahezu 50 % kleiner als der generierte Code seines Vorgängers.

Verwendeter Compiler	Größe des src/- Ordners
ViewEngine	92Kb
Ivy	50Kb

Wirft man einen Blick auf die generierten Dateien, kann man zudem sehen, dass die ViewEngine für unsere *AppComponent* ganze vier verschiedene Dateien (zwei JavaScript- und zwei JSON-Dateien) generiert, wohingegen Ivy lediglich nur eine einzige JavaScript Datei erstellt.

Mit der ViewEngine werden aus einer Komponente zwei JavaScript-Dateien und zwei JSON-Dateien generiert



Mit Ivy wird lediglich nur noch eine einzige JavaScript-Datei generiert

In der *app.component.js*, welche von der ViewEngine generiert wurde, befindet sich hierbei lediglich der Controller-Code unserer Komponente. Sämtliche Anleitungen zum Aufbau der HTML-Struktur finden wir in der *app.component.ngfactory.js*. Die beiden JSON-Dateien geben uns darüber hinaus noch Aufschluss, welche Abhängigkeiten unsere Komponente zusätzlich benötigt, um korrekt in dem Web-Browser dargestellt werden zu können.

Wenn wir uns den Ivy-generierten Code ansehen, stellen wir fest, dass funktionaler Code, sowie die Anleitung zur Erstellung des Templates innerhalb derselben JavaScript-Datei zu finden sind.

```
AppComponent.ɵfac = function AppComponent_Factory(t) { return new (t || AppComponent)(); };
AppComponent.ɵcmp = core.ɵɵdefineComponent({
    type: AppComponent,
    selectors: [["app-root"]],
    decls: 2,
    vars: 1,
    template: function AppComponent_Template(renderflags, ctx) {
        if (renderflags & RenderFlags.INIT) {
            core.ɵɵelementStart(0, "div");
            core.ɵɵtext(1);
            core.ɵɵelementEnd();
        }
        if (renderflags & RenderFlags.UPDATE) {
            core.ɵɵadvance(1);
            core.ɵɵtextInterpolate1(" Welcome to ", ctx.title, "!\n");
        }
    },
    styles: [""]
});
```

Zum Code →

Mit Ivy landet Controller-Code und Code für das Rendern des HTML-Baumes in einer Datei.

Mit Ivy wurden sogenannte statische Attribute eingeführt, welche beispielsweise eine Komponente, eine Direktive oder eine Pipe mit ihren Metadaten und ihrem Aufbau beschreiben. In unserem Listing ist es das *ɵcmp*-Attribut der AppComponent. Dieses Attribut erwartet eine Komponenten-Definition, welche man mit dem Aufruf der Funktion *ɵɵdefineComponent* des Angular-Core Frameworks erhält. Werfen wir einen Blick auf das Objekt mit dem die Funktion aufgerufen wird, sehen wir die *AppComponent_Template*-Funktion, welche für den Aufbau unseres HTML-Templates verantwortlich ist.

Dabei werden weitere sogenannte Template-Funktionen des Angular-Frameworks aufgerufen, wie beispielsweise *ɵɵelementStart*, welches unseren div-HTML-Knoten erstellt oder *ɵɵtext*, das für die Erstellung eines Text-Knoten in unserem Document Object Model sorgt.

Ebenso können wir erkennen, dass unsere Template-Funktion auf zwei unterschiedliche Zustände reagiert: *Create* und *Update*.

Während bei dem ersten Zustand das gesamte HTML aufgebaut wird, ist der *Update*-Zustand für die String-Interpolation, also dem eigentlichen Einfügen des Wertes der *title*-Variable in das HTML, zuständig, sowie die gesamte Change Detection.

Des Weiteren ist auffällig, dass Zeichen *ɵ* (griechisches Theta) Teil des Funktion-Namens ist. Dies soll lediglich aussagen, dass es sich um eine private API des Core Frameworks handelt und sich im Laufe der nächsten Release Candidates noch ändern kann und deshalb die Nutzung dieser API mit Vorsicht zu genießen ist.

Diese API wurde mit Ivy eingeführt, doch welchen Vorteil bezüglich der Bundle-Size erhalten wir nun durch sie?

Um diese Frage beantworten zu können, müssen wir uns vorerst noch einmal ansehen, wie die HTML-Struktur mithilfe der ViewEngine bisher gerendert wurde. Betrachten wir ein ganz simples HTML-Template und den von der ViewEngine kompilierten JavaScript-Code, sehen wir innerhalb der JavaScript-Datei einen Funktionsaufruf mit einem Array als Übergabeparameter. Dieses Array beinhaltet mehrere sogenannte Template-Instruktionen. In der Instruktion *elementRef* können wir beispielsweise den genauen Aufbau des HTML-Knotens *div* einsehen.

```
<div class="blue">
  Hello NG-DE
</div>
```

```
viewDef([
  elementRef(0, 'div', ['class', 'blue'], ...);
  textDef('Hello NG-DE', ...);
])
```

Beispiel eines HTML-Template und dem dazugehörige JavaScript-Source-Code, welcher von der ViewEngine generiert wurde.

Sehen wir uns nun einmal an, wie das Angular-Framework bisher mithilfe dieser Template-Instruktionen den DOM-Baum rendert. Hier sehen wir den sehr vereinfachten Code der Core Library:

```
function createViewNodes(view: ViewDef) {
 ...
 view.nodes.forEach ((node => {
   case 1: /* TypeElement */
      createElement(view, rederHost, nodeDef);
      break;
   case 2: /* TypeText */
      createText(view, rederHost, nodeDef);
      break;
   case 11: /* TypePipe */
      createPipe(view, nodeDef);
      break;
 })
}
```

Zum Code →

Auszug aus dem Angular-Framework zum Rendern der Komponenten

Das heißt: für jede Instruktion innerhalb eines Template-Codes wird ein einfaches switch-case durchlaufen und die entsprechende Funktion für das HTML-Rendering aufgerufen. Dies ist ein Design-Ansatz der häufig bei Interpretern eingesetzt wird und womöglich deshalb auch hier dementsprechend implementiert wurde.

Der größte Nachteil bei diesem Ansatz ist jedoch, dass zum Zeitpunkt der Kompilierung nicht festgestellt werden kann, welche dieser Funktionen für das Rendern eines HTML-Knotens aufgerufen werden und welche nicht.

Konkret heißt das, dass in unserem Beispiel der gesamte JavaScript-Source Code für die Funktion ***createPipe*** in das Bundle kompiliert und somit auch an den Web-Browser ausgeliefert wird, obwohl wir diese Funktion niemals benutzen.

In diesem sehr vereinfachten Beispiel gibt es nur drei Cases, insgesamt sind es allerdings weit über 15 verschiedene Möglichkeiten und Funktionen, die eventuell mit ausgeliefert werden, obwohl wir diese eigentlich nicht benötigen (für einen genaueren Blick in die Funktion ***createViewNodes*** kann diese eingesehen werden, indem man das Angular-Projekt mit ***ng build*** kompiliert. Die Funktion befindet sich daraufhin in der ***vendor-xxx.js*** Datei).

Die ViewEngine folgt demnach nicht dem Prinzip des sogenannten „Tree-Shaking" (auch „Dead Code Elimination" genannt), welches besagt, dass Code, der nicht benutzt — also nicht aufgerufen — wird, nicht mit in das Bundle kompiliert wird.

Und wie ist es bei Ivy? Genau hier kommt die neue API ins Spiel: mithilfe der neuen Engine rufen die Komponenten nun selbst direkt den Source-Code zu dem Rendern der HTML-Knoten auf.

Das bedeutet, dass bereits zum Zeitpunkt der Kompilierung genau bestimmt werden kann, welche Funktionen eine Komponente benötigt, um gerendert zu werden. Daraus folgt wiederum, dass Funktionen, die von keiner Komponente aufgerufen werden, aus dem Angular-Core-Bundle entfernt werden können.

Definiert man also in der ***app.component.html***-Datei lediglich HTML- und Text-Knoten, wie in unserem Einführungsbeispiel, ist der Vergleich der Bundle-Sizes zwischen dem Vorgänger ViewEngine und Ivy signifikant. Man muss allerdings auch erwähnen, dass die Neuerung

dieser Dead-Code-Eliminierung hierbei lediglich Template-Source-Code betreffen. Modul-Source-Code, wie er beispielsweise in dem *ReactiveFormModule* oder dem *HttpClientModule* vorkommt, wurde zuvor bereits nach diesem Prinzip reduziert.

Schnellere Rebuilds von Angular-Apps

Ein weiteres Feature, welches mit Ivy nun ausgeliefert wird, ist das schnellere Rebuilden von unserer Angular-Applikation. Dies wird ermöglicht durch die separate Kompilierung jeder einzelnen Datei ohne weitere Abhängigkeiten. Was vorerst recht logisch erscheint, sah in dem von der ViewEngine-generierten Code allerdings noch anders aus. Zu Demonstrationszwecken haben wir hierfür unser HTML-Template mit einer einfachen *ngIf*-Direktive erweitert und das Projekt erneut kompiliert.

Hier ein Auszug des kompilierten JavaScript Codes:

```
1  export function View_AppComponent_0(_l) {
2    return i1.ɵvid(0,[
3     ...
4     i1.ɵdid(2, 16384, null, 0, i2.NgIf, [i1.ViewContainerRef, i1.TemplateRef],{ngIf: [0, "ngIf"]}, null)
5     ...
6    ])
7    ...
8  }
```

Zum Code →

Diese Komponente beinhaltet auch die Information der Abhängigkeiten der ngIf-Direktive

Neben der bereits bekannten Template-Instruktion für das Rendern des *div*-HTML-Knotens, sehen wir nun zusätzlich eine Definition der ngIf-Direktive. Allerdings erhält diese Instruktion noch einen weiteren Parameter: *[i1.ViewContainerRef, i1.templateRef]*. Dieses Array ist allerdings dabei keine Abhängigkeit der Komponente direkt, sondern der *ngIf*-Direktive.

Demzufolge beinhaltet der Komponenten-Code nicht nur seine eigene Abhängigkeit, sondern auch die Abhängigkeiten ihrer Abhängigkeiten. Die Auswirkungen bei dem Ändern der Abhängigkeiten der benutzten Direktive sind hierbei schnell ersichtlich:

Sämtliche Komponenten, die diese Direktive benutzen, müssen ebenfalls neu kompiliert werden. Dabei spricht man von einer sogenannten globalen Kompilierung.

Hier sehen wir nun das Ivy-kompilierte Äquivalent:

```
1           if (renderflags & RenderFlags.UPDATE) {
2               core.ɵɵadvance(1);
3               core.ɵɵproperty("ngIf", ctx.title);
4               core.ɵɵadvance(1);
5               core.ɵɵtextInterpolate1(" Welcome to ", ctx.title, "!\n");
6           }
```

Zum Code →

Lokalitätsprinzip: Die Komponente kennt nur noch ihre direkten Abhängigkeiten

Hierbei werden lediglich die direkten lokalen Abhängigkeiten der Komponente deklariert: Es handelt sich um das sogenannte Lokalitätsprinzip. Die neue API liefert dementsprechend auch wieder eine Funktion für die Definierung der Direktive: *ɵɵdefineDirective*. Diese wiederum beinhaltet sämtliche Informationen, die ein *ngIf* benötigt, um dargestellt zu werden (wie beispielsweise den *ViewContainerRef* und *templateRef*).

Durch die klare Abtrennung der Abhängigkeiten mit Hilfe dieser sauberstrukturierten API, ist es von nun an möglich, nur noch die Dateien neu zu kompilieren, welche sich tatsächlich verändert haben.

Doch das ist noch nicht alles: durch die klare Abtrennung können externe Libraries bereits Ahead-of-time kompiliert und in unsere Angular-Applikation importiert werden.

Zusätzlich erwähnenswert ist noch, dass neben dem Lokalitätsprinzip natürlich auch das Kompilieren zu einer einzigen JavaScript-Datei dem schnelleren Rebuilden beiträgt. Der Compiler muss nicht mehr vier Dateien und deren Referenzen zueinander erzeugen.

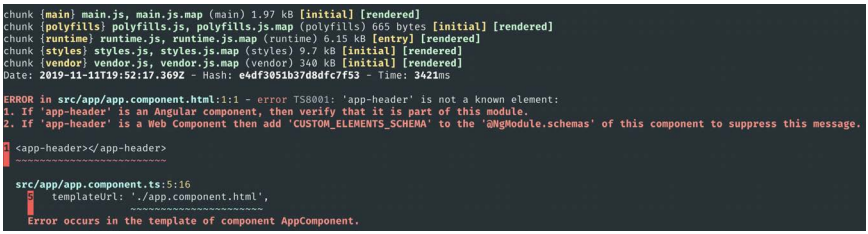
Verbessertes Debugging

Die Reduzierung der Bundle-Größe und das schnellere Rebuilden der Angular-Applikation waren zwar der Hauptfokus des Angular-Teams für die neue Rendering Engine, erwähnenswert ist zusätzlich allerdings auch noch die Verbesserung des Debuggings der Angular-Anwendung:

Die Templates der Komponenten werden dabei im Stack Trace des Browsers nun mehr sichtbar, sodass Fehlermeldungen eindeutiger sind und sich das Debugging durch Breakpoints im Template einfacher gestaltet.

Kämpfte man sich dabei bisher durch einen sehr komplexen und verworrenen ViewEngine-generierten JavaScript Code, ist der gesamte Funktions-Aufruf Stacktrace dank der neu eingeführten API leichter lesbar und vor allem nachvollziehbar.

Sämtliche JavaScript-Fehlermeldungen, für die man bisher seine Browser-Developer-Tools öffnen musste, werden seit Angular 9 zusätzlich ebenso in der Kommandozeile sichtbar, nachdem man seine Angular-Applikation mit dem Befehl *ng serve* gestartet hat.



Verbesserte Konsole-Ausgaben bei dem Kompilieren von Angular Applikationen

Mit Angular 9 und der neuen Rendering Engine wird also alles besser? Der eine oder andere wird sich womöglich noch die Frage stellen, wie es dies bezüglich um all die Abhängigkeiten einer Angular-Applikation steht. Viele Third-Party Libraries lassen sich dabei noch nicht mit Ivy kompilieren und erscheinen inkompatibel zu der neuen Engine. Aber auch hierfür hat das Angular-Team eine Lösung gefunden.

Eine radikale Umstellung der gesamten Rendering-Engine bringt natürlich vielerlei Herausforderungen mit sich. So mussten die Angular-Entwickler dafür sorgen, den Übergang so sanft wie möglich zu gestalten. Backwards-Kompatibilität war hierbei das Ziel, und so bietet Angular zwei unterschiedliche Compiler an:

- *Ngtsc*
- *Ngcc*

Der Ngtsc ist hierbei der bereits beschriebene Ivy Compiler, welcher den Angular-Code, der bereits in einem Ivy-kompatiblen Format vorliegt, kompiliert.

Interessant ist allerding der Ngcc, der sogenannte Angular-Kompatibilitäts-Compiler. Dieser sorgt dafür, das Libraries innerhalb des *node_modules* Ordners, welche nicht mit Ivy kompiliert wurden, so verändert werden, dass diese Ivy-kompatibel sind.

Beispielsweise wandelt er sämtliche Decorators (@Pipe, @Component, @NgModule etc) in die entsprechenden statischen Attribute: *definePipe*, *defineComponent*, *defineDirective* um. Dies ermöglicht die Nutzung von „legacy" Projekten innerhalb eines mit Ivy-kompilierten Projekts.

Zusätzlich hat das Angular-Team mit dem ersten Release Candidate offiziell dazu aufgerufen, aktiv an der Kontribution von Ivy teilzunehmen. Das gesamte Team arbeitet daran, die Migration nach Ivy so schnell und vor allem, so stabil wie möglich zu gestalten.

So hat das Team mittlerweile ein Feature in die Angular-CLI eingebaut, welches erlaubt, Statistiken über die Nutzung und Kompilierung von Angular-Projekten zu erheben. Dies dient lediglich der Verbesserung von Ivy und Angular selbst.

```
? Would you like to share anonymous usage data about this project with the Angular Team at
Google under Google's Privacy Policy at https://policies.google.com/privacy? For more
details and how to change this setting, see http://angular.io/analytics. Yes

Thank you for sharing anonymous usage data. Would you change your mind, the following
command will disable this feature entirely:

    ng analytics project off
```

Helfen Sie dem Angular-Team, indem Sie Statistiken über Ihr Angular-Projekt mit dem Team teilen

Natürlich ist diese Erhebung vollkommen freiwillig und kann jederzeit auch wieder deaktiviert werden.

Das eigene Projekt auf Ivy zu migrieren, birgt sicherlich noch einige Hürden, die erst nach und nach von dem Team behoben werden können. Dabei hat unter den Angular-Entwicklern Ivy allerdings höchste Priorität. Umso wichtiger ist es, Probleme und Herausforderungen, auf die man während der Umstellung stößt, dem Angular-Projekt als 'Issue' zu melden.

Zusammenfassung und Ausblick
Wie wir gesehen haben, bringt uns die Umstellung auf die neue Rendering Engine eine Reihe von Vorteilen:

Die Größe unseres Bundles, welches wir an den Web-Browser ausliefern, kann hierbei signifikant kleiner sein als noch zu Zeiten der ViewEngine. Das liegt, wie wir gelernt haben, an der Möglichkeit, nicht benutzten Code zur Kreierung unserer HTML-Elemente nicht mitliefern zu müssen, da mit Hilfe von Ivy bereits zum Kompilierung-Zeitpunkt feststeht, welche Framework-Funktionen unsere Komponente benötigt, und welche nicht.

Dank der neuen API und den statischen Attributen zur Definition einer Komponente, einer Direktive oder einer Pipe müssen mit Ivy nun nicht mehr Abhängigkeiten von Abhängigkeiten innerhalb einer Komponente deklariert werden. Eine Komponente beinhaltet nur noch die Informationen ihrer direkten Abhängigkeiten und muss lediglich nur noch kompiliert werden, wenn sich etwas an der Komponente selbst verändert. Dieses sogenannte Lokalitätsprinzip ermöglicht das schnellere Rebuilden der Angular-Applikation, da nicht mehr das komplette Projekt global gebaut werden muss. Auch externe Bibliotheken können dabei bereits optimiert und vorkompiliert importiert werden.

Neben den verbesserten Debugging-Möglichkeiten haben wir auch den Angular Kompatibilitäts Compiler (ngcc) kennengelernt. Dieser hilft uns, Legacy-Code in unserem *node_modules* Ordner in ein für Ivy-lesbares Format zu überführen. Damit wird uns ermöglicht, selbst ältere Anwendungen mit dem neuen Renderer zu nutzen.

Während dieser Artikel nur eine kleine Übersicht der neuen Angular-Engine geben konnte, verbirgt sich hinter Ivy noch einiges mehr:

Mit der neuen API kann man beispielsweise auch ganz einfach Komponenten zu jedem Zeitpunkt der Angular-Anwendung dynamisch erstellen und laden: Das Prinzip der sogenannten High Order Components.

Ebenso ist es vorstellbar, dass mit Ivy in Zukunft die Angular-Entwicklung vollkommen von dem Konzept der sogenannten Angular-Module losgelöst sein wird. Hierbei verweise ich auf die Artikel-Serie von Manfred Steyer „Architecture with Ivy: A possible future without Angular Modules", welche weiterführend auf High Order Components und das Entwickeln ohne Angular-Module eingeht und dies erläutert.

Der Autor

David Würfel

David Würfel ist Senior Design Engineer beim User Experience Dienstleister Centigrade GmbH am Standort Saarbrücken. Als Field Lead leitet er den Bereich der Webentwicklung. Er unterstützt u.a. große Kunden aus der Industrie beim Aufbau von Control Bibliotheken und Design Systemen oder bei der Umsetzung von ästhetischen und benutzerfreundlichen Applikationen zumeist mit Angular. Sein Wissen teilt David gerne intern wie extern in Reviews, Trainings oder als Speaker auf Fachkonferenzen. Er ist aktiver Unterstützer lokaler Meetups und bringt sich gerne in die Community ein. In seiner Freizeit entspannt er handwerklich, kreativ beim Bemalen kleiner Plastikfiguren.

* Die Illustrationen und der Quellcode der Beispielapp **NgCompanion** sind im Rahmen meiner Arbeit bei Centigrade GmbH entstanden und wurden bereits in einem Vortrag auf der Web Developer Conference 2019 vorgestellt.

Von vorne bis hinten: Angular ♥ NestJS

Mittlerweile sind viele von uns sehr vertraut mit dem Entwickeln clientseitiger Apps mit Angular und TypeScript. Bauen wir diese im Handumdrehen, brauchen wir irgendwann wahrscheinlich eine serverseitige API, um mit echten Daten umzugehen. Es wäre nur natürlich, im Backend die gleichen Sprachen zu nutzen wie im Frontend. Also entscheiden wir uns z. B. für Node.js. Doch wenn wir reines Node oder Bibliotheken wie Express.js nutzen, betreten wir Frontend-Entwickler oft unbekanntes Terrain. Uns fehlen Idiome und strukturelle Muster.

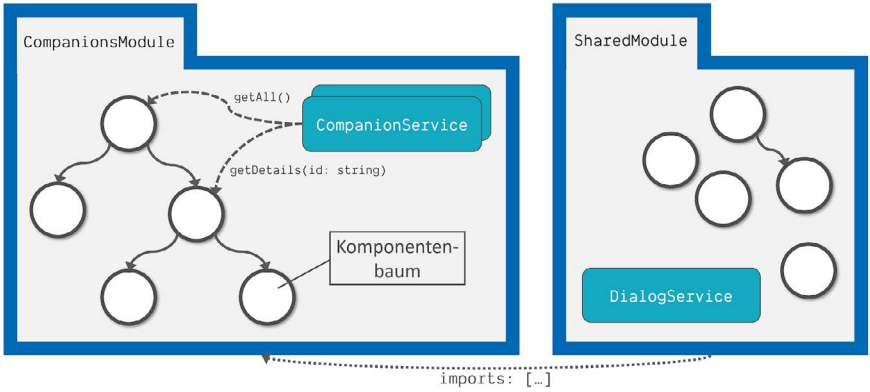
Statt im Dickicht von zahlreichen Low-Level Bibliotheken in Verbindung mit Express zu versinken, wollen wir uns NestJS anschauen. Dieses Framework ermöglicht es uns, effiziente, zuverlässige und skalierbare serverseitige Anwendungen zu erstellen. Bemerkenswert daran ist, dass es sich ähnlicher Konzepte bedient wie Angular.

Im Folgenden schauen wir uns an, wie eine Projektstruktur mit Angular-Front- und Nest-Backend aussehen kann und schreiben unsere ersten API-Routen. Weil es so einfach ist, erstellen wir nebenbei eine API-Dokumentation und werfen einen Blick auf weitere nützliche Plugins. Am Ende sollten wir gerüstet sein, eine Full-Stack-Webanwendung durch die Symbiose von Angular und Nest umzusetzen.

Angular kurz und bündig

Was wir häufig an Angular schätzen, ist seine klare Struktur und die festen Wege, wie wir Anwendungen schreiben. In einem gut aufgesetzten Angular-Open-Source-Projekt findet man sich schnell zurecht, man kennt die Bestandteile, aus denen die Applikation besteht und weiß, welche Wechselwirkung es zwischen ihnen gibt.

Kurz gefasst fühlt sich das wie folgt an: Mit der Angular CLI können wir zügig ein gesamtes Projekt oder später Artefakte samt dem dazugehörigen Boilerplate-Code erzeugen. Unsere Benutzeroberfläche bauen wir zusammen aus einem Baum von vielen kleinen, eher controlartigen Komponenten, die in einigen wenigen seitenartigen Komponenten miteinander verknüpft werden. Die Seiten bedienen sich sogenannter Services, die wiederum für die Datenverarbeitung und Anwendungskommunikation zuständig sind. Zuletzt ordnen wir diese Artefakte in verschiedene Module ein, die jeweils zusammengehörige Pakete bilden und jeweils eigene Anwendungsbereiche abdecken.

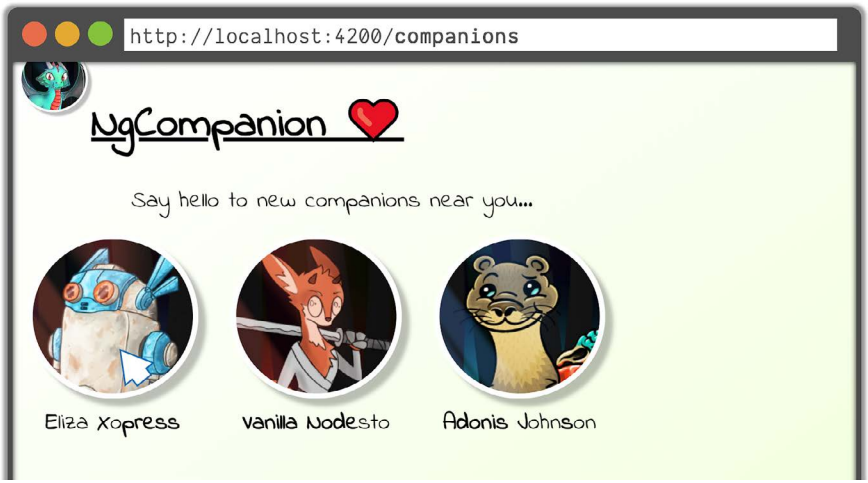


Eine typische Angular-Anwendungsarchitektur: Aufgeteilt in Module liegen unsere verschiedenen Komponentenbäume, die die Oberfläche entstehen lassen und ihre Daten über Services beziehen.

Dabei nutzen wir alle uns zur Verfügung stehenden Werkzeuge wie TypeScript, Linting (die statische Analyse unseres Quellcodes, die uns hilft, gewisse Pattern einzuhalten), das Live-Reload-Feature unseres Browsers und die hilfreiche Testinfrastruktur, die uns die CLI bietet.

Beispiel: NgCompanion*

Stellen wir uns vor, wir wollten eine Social Media Plattform namens *NgCompanion* entwickeln. In dieser meldet man sich an, hinterlässt ein paar Profilinformationen und nutzt die Applikation, um Freunde und Gleichgesinnte in der Umgebung zu finden.



Screenshot unserer Client-App NgCompanion. Wir sind angemeldet als *Andre Gular* und sehen in der Übersicht eine Reihe interessanter Bekannte in der Nähe.

Unser Frontend haben wir mit Angular bereits aufgesetzt und entwickelt. Darin befinden sich verschiedene Komponenten, beispielsweise zum Anzeigen unseres Benutzerprofils, der Darstellung eines potenziellen Freundes oder die Seiten-Komponente, die die Liste der Freunde rendert. Diese Listenübersicht bezieht ihre Daten letztlich über einen Angular-Service *CompanionsService*. Innerhalb dieses Service kann man den *HttpService* nutzen, um über eine REST API an diese Daten aus dem Backend zu gelangen:

```
1  export class CompanionsService {
2
3    constructor(private http: HttpClient) {}
4
5    public getAll(): Observable<Companion[]> {
6      const url = `${BASE_URL}/companions`;
7      return this.http.get<Companion[]>(url);
8    }
9
10   // ...
11 }
```

Zum Code →

Angular-Service, der über eine REST API die interessanten Bekannten in der Nähe, zur Darstellung in der Client-App, zurückliefert.

Zur Umsetzung dieser REST API wollen wir nun die dazugehörige Node.js Backend-Schnittstelle entwickeln. Dies könnten wir mit der Node Bibliothek Express tun. Damit sind wir sehr flexibel, müssten uns die Bibliothek über *npm* installieren und könnten dann direkt loslegen. Standardmäßig stehen uns dann aber nicht direkt all die Tools wie TypeScript, Live-Reload oder eine klar vorgegebene Struktur zur Verfügung. Best Practices sind zwar vorhanden, müssen wir uns jedoch selbst erst mühsam erarbeiten

und in unserem Projekt umsetzen. Für Frontend-Entwickler, die seltener Backend-Systeme aufsetzen, eine ziemlich große Einstiegshürde. Könnte das nicht etwas zugänglicher sein?

Vor- und Nachteile beim Einsatz eines Frameworks wie Express.js zur Umsetzung einer Node.js Backend-Schnittstelle:

- Große Flexibilität
- Wenig Abhängigkeiten
- Schmales, unkompliziertes Framework
- Wenig vorgegebene Struktur
- Viel manuelle Konfiguration notwendig
- Standardmäßig kein TypeScript
- Live-Reload muss erst konfiguriert werden
- Architektur ist dem Anwendungsentwickler überlassen
- Wenig Wiedererkennungswert: Jede Express-Applikation mag anders aussehen

NestJS

Genau hier kommt NestJS, kurz Nest, ins Spiel. Laut Webseite „ein progressives Node.js-Framework für die Erstellung effizienter, zuverlässiger und skalierbarer serverseitiger Anwendungen". Es wurde von Kamil Mysliwiec ins Leben gerufen, ist unter MIT Lizenz Open Source, hat zahlreiche Unterstützer und wurde 2019 als das auf GitHub populärste Node.js-Framework gekürt. Zum Zeitpunkt dieses Artikels ist das Framework in Version 7 erschienen. Es verfügt über eine ausgezeichnete Dokumentation, die uns auch über dessen Kernfunktionalität informieren kann. Unter der Haube nutzt Nest standardmäßig Express, kann aber hier

auch andere Bibliotheken wie beispielsweise Fastify einsetzen. Es liefert eine vorgeschriebene Applikationsarchitektur für unser Backend, die sich aus ähnlichen Artefakte wie Angular zusammensetzen und mit Dekoratoren annotiert wird.

Aufsetzen des Backend-Projekts

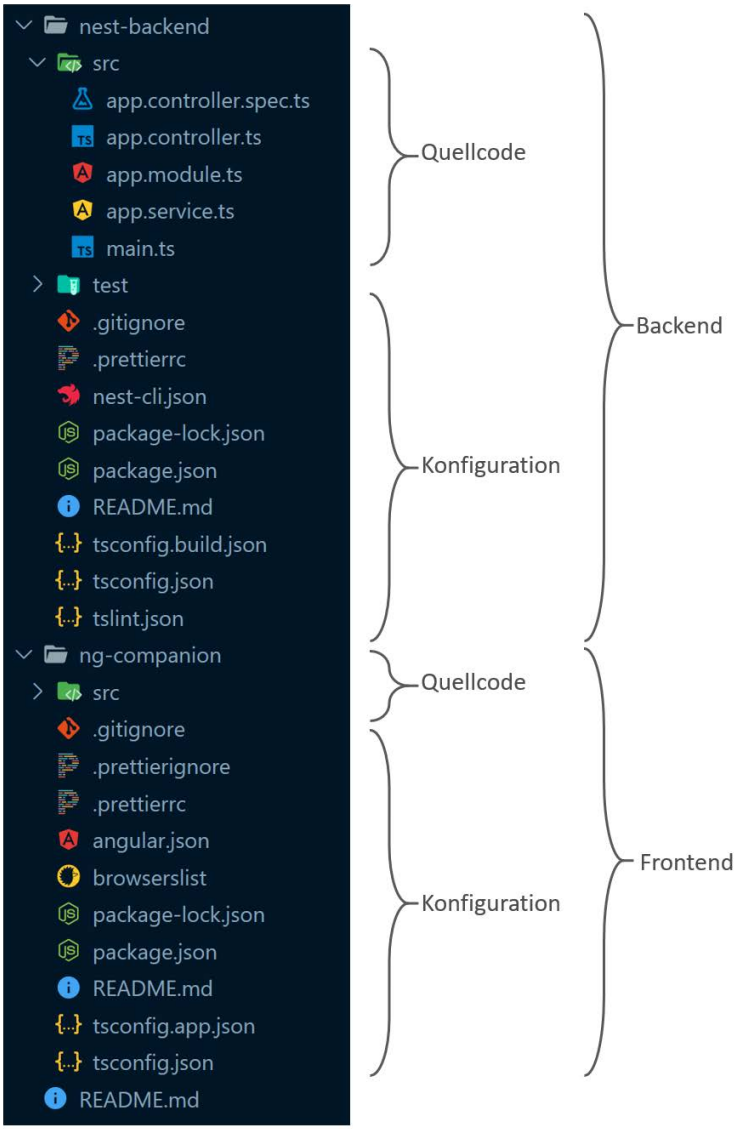
Auch Nest verfügt über eine CLI, mit der wir unsere Backend-Anwendung erstellen können. Es bietet sich an, sowohl das Front- als auch das Backend in einem gemeinsamen Repository, aber mit eigenständigen Abhängigkeits-Definitionen, eingetragen in der ***package.json,*** zu erstellen. So können wir später gemeinsam genutzte Datentransfer-Objekte im Wurzelverzeichnis anlegen und zwischen beiden Welten teilen.

Mit folgendem Befehl erstellen wir uns ein neues Nest-Projekt, welches wir wie vorgeschlagen neben unserem Frontend-Projekt in einem eigenen ***backend*** Ordner verorten:

```
> npx @nestjs/cli new ng-companion-backend
```

*Wir nutzen das npx Kommando, dass uns seit npm Version 5.2.0 erlaubt, Pakete ohne die Notwendigkeit einer globalen Installation zu nutzen. Da danach Nest und seine CLI selbst als Abhängigkeit im Projekt vorhanden ist, brauchen wir zu keinem eine globale Installation.

Öffnen wir nun den neu erstellten Ordner und werfen einen Blick in die dort erstellten Dateien und die vorgegebene Ordnerstruktur, so könnten uns einige Parallelen in Bezug auf die Dateistruktur einer Angular-Applikation auffallen, wie beispielsweise die ***nest-cli.json*** als zentrale Konfigurationsdatei ähnlich zur ***angular.json***.



Screenshot von Front- und Backend im Monorepo.
Initiale Ordner- und Dateistruktur einer Nest-Applikation.

Mit folgendem Befehl starten wir unser Backend im Watch-Modus und können das Backend standardmäßig über die URL http://localhost:3000 erreichen:

```
> npm run start:dev
```

Startet das Backend durch :dev im Watch-Modus und liefert es auf http://localhost:3000 aus.

Bestandteile einer Nest-Applikation

Schauen wir uns die Inhalte des *src* Ordners an, sehen wir die typischen Bestandteile einer Nest-Applikation. Der initiale Einstiegspunkt ist die Datei *main.ts*. Hier findet das Bootstrapping des Backends statt. Diese Datei ist auch der Ort, an dem grundlegende Änderungen an der Plattformkonfiguration vorgenommen werden, wenn z. B. statt Express Fastify als unterliegendes Framework genutzt werden soll.

```ts
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule); // Hier könnte man die Plattform von Express auf Fastify wechseln
  app.enableCors();
  await app.listen(3000);
}

bootstrap();
```

Zum Code →

Inhalt der main.ts zum Start des Nest-Backends. Die Konfiguration der Plattform, das Vornehmen zusätzlicher Einstellungen wie CORS, das Zuweisen des Ports oder das Hinzufügen von optionalen Plugins findet hier statt.

Neben diesem Einstiegspunkt findet man eine Reihe von Dateien mit dem Präfix app im Quellcode-Verzeichnis. Hier erkennt man alle grundlegenden Bausteine einer Nest-Applikation, die sehr verwandt zu den Bausteinen eines Angular Frontends sind.

Controller (app.controller.ts)

Ähnlich der ***Components*** in Angular sind ***Controller*** in Nest die Einstiegspunkte zur Verarbeitung verschiedener URL-Routen, sozusagen die äußerste Schnittstelle der Applikation zum Benutzer.

Service (app.service.ts)

Während die Controller nur für die erste Verarbeitung der von außen erreichbaren URL-Routen zuständig sind, leiten sie anschließend weiter zu den verschiedenen ***Services***, die wiederum den Kern der Applikationslogik enthalten sollten. Ähnlich wie auch Services in Angular, können Controller Services mit Hilfe der Dependency Injection über den Konstruktor anfordern.

Module (app.module.ts)

Zuletzt teilt man wie bei Angular in einem Nest-Backend verschiedene zusammengehörige Bereiche in Module auf, um so eine wartungsfähige, austausch- und erweiterbare Gesamtarchitektur aufzubauen. In den Modulen werden dafür notwendige Controller und Services registriert.

Implementierung der ersten Backend Logik

Wir möchten nun die für unser Frontend notwendige GET-Routen implementieren, die uns die Liste aller Freunde in der Nähe sowie Details zu einem bestimmten Freund, den wir per ID auswählen, liefern:

- ***http://localhost:3000/companions***
  liefert die Liste aller verfügbaren Freunde
- ***http://localhost:3000/companions/:id***
  liefert Details zu dem Freund einer bestimmten ID

Controller

Dazu erstellen wir uns mit Hilfe der Nest CLI einen Controller zur Behandlung dieser Routen. Die CLI fügt diesen Controller auch automatisch dem nächstgelegene Modul hinzu und erstellt ebenfalls einen Unit-Test mit der Endung ***.spec.ts***, wie wir es aus Angular kennen:

```
> nest generate controller companions
> nest g co <controller-name> # Kurzform
```

Erstellen eines Nest-Controllers mit der CLI in Lang- oder Kurzform.

Analog zu Komponenten in Angular (***@Component*** Decorator), ist der Controller mit dem Decorator ***@Controller*** versehen. Hier sollte der Name der Route (***companions***) angegeben werden. Wir definieren nun über den ***@Get*** Decorator jeweils eine Methode, die die Routen-Anfragen entgegennimmt und verarbeitet:

```ts
import { Companion, CompanionDetails } from '@interfaces';
import { Controller, Get, Param } from '@nestjs/common';
import { ApiOperation, ApiResponse, ApiUseTags } from '@nestjs/swagger';
import { CompanionsService } from '../../services/companions/companions.service';

@Controller('companions')
export class CompanionsController {

  @Get()
  async getAll(): Promise<Companion[]> { // Hier kann man ebenfalls Companion[] oder Observable<Companion[]> zurückliefern.
    // ...
  }

  @Get(':id')
  async getCompanionDetails(@Param('id') id: string): Promise<CompanionDetails> {
    // ...
  }
}
```

Zum Code →

Grundgerüst der ersten zwei GET-Routen zur Verarbeitung der Routen **/companions** und **/companions/:id.** An Stelle von Promises, können auch Observables verwendet werden.

Damit wir bei der zweiten Route Zugriff auf die in der URL übergebene ID bekommen, müssen wir lediglich den Parameternamen im *@Get* Decorator angeben und den entsprechenden Methodenparameter mit *@Param* markieren. Nest kann an dieser Stelle sowohl mit synchronen wie auch asynchronen Methoden auf Basis von Promises oder Observables umgehen.

Services

Um nun die eigentlichen Daten beispielsweise aus verschiedenen Quellen, wie z. B. einer Datenbank zu laden, zu kombinieren und aufzubereiten, lagern wir diese Kernfunktionalität in Services aus, die wiederum der Controller nutzt. Wir erstellen auch diesen mit der CLI:

```
> nest generate service companions
> nest g s <service-name> # Kurzform
```

Erstellen eines Nest Services mit der CLI in Lang- oder Kurzform.

Ein solcher Service muss wie bei Angular mit dem Decorator *@Injectable* annotiert sein. Ein Service kann wiederum andere Services beinhalten und beliebige komplexe Logik implementieren. An unserer Stelle gehen wir davon aus, dass wir eine existierende Datenbank-Schnittstelle nutzen und von dieser entsprechende Informationen abfragen.

```
1  @Injectable()
2  export class CompanionsService {
3
4    // … hier könnte z.B. eine Datenbankanbindung stattfinden
5
6    async findAll(): Promise<Companion[]> {
7      return await this.db.find(...).exec();
8    }
9
10   async findDetailsById(id: string): Promise<CompanionDetails> {
11     const companionDetails = await this.db.findOne({ id }).exec();
12     return companionDetails;
13   }
14 }
```

Zum Code →

Angedeutete Service-Implementierung zur Datenbeschaffung unserer Companions.

Der Service kann daraufhin in Modulen bereitgestellt und von Controllern angefragt werden. Dazu genügt es, sich den Service über den Konstruktor injizieren zu lassen. Ähnlich zu Container-Komponenten in Angular erkennt man, dass der Controller in Nest nur eine sehr schmale Schicht ist, der Nutzerdaten entgegennimmt, diese für die Weiterverarbeitung aufbereitet und die eigentliche Logik innerhalb verschiedener Services stattfindet.

```
export class CompanionsController {
  constructor(private service: CompanionsService) {}

  @Get()
  async getAll(): Promise<Companion[]> {
    return this.service.findAll();
  }

  @Get(':id')
  async getCompanionDetails(@Param('id') id: string): Promise<CcompanionDetails> {
    return this.service.findDetailsById(id);
  }
}
```

Zum Code →

Injizierung des zuvor implementierten CompanionsService und dessen Verwendung innerhalb des Controllers.

Module

Zuletzt müssen Controller und Services noch in einem Modul registriert werden. Dies wurde durch die CLI bereits in dem aktuell einzigen Modul, dem *app.module*, vorgenommen. Es könnte jedoch sinnvoll sein, dass wir unseren Bereich *Companions* in ein eigenes gleichnamiges Modul verschieben. Auch dieses kann mit der CLI erstellt werden:

```
> nest generate module companions
> nest g mo <service-name> # Kurzform
```

Erstellen eines Nest-Moduls mit der CLI in Lang- oder Kurzform.

In dieses Modul platzieren wir in unserem Fall unseren Controller sowie den Service.

```ts
import { Module } from '@nestjs/common';
import { CompanionsController } from './controllers/companions/companions.controller';
import { CompanionsService } from './services/companions/companions.service';

@Module({
  imports: [/* Hier könnte das Datenbankmodul genutzt werden */])],
  controllers: [CompanionsController],
  providers: [CompanionsService],
})
export class CompanionsModule {}
```

Zum Code →

Das **CompanionsModule** enthält dazugehörige Controller und Services.

Einzelne Module können dann wiederum auf der obersten App-Ebene mit in die Applikation aufgenommen werden. So könnte man neben dem ***CompanionsModule*** noch viele weitere andersartige Teilbereiche des Backends unabhängig davon implementieren.

```ts
import { Module } from '@nestjs/common';
import { CompanionsModule } from './companions/companions.module';

@Module({
  imports: [
    CompanionsModule,
    // Viele weitere Module denkbar
  ],
})
export class AppModule {}
```

Zum Code →

Das Root-Modul AppModule bündelt alle sogenannten Feature-Module, wie beispielsweise unser CompanionsModule.

POST-Route

Haben wir aktuell nur Daten aus dem Backend abgefragt, so ist es auch möglich, Daten an das Backend zu senden. Möchten wir aus dem Frontend heraus zum Beispiel die Kontaktaufnahme mit einem ausgewählten Freund initiieren, so können wir hierzu einen POST-Request implementieren. Dazu müssen wir den *@Post* Decorator an einer Methode verwenden und geben als Parameter das mit *@Body* annotierte Datentransfer-Objekt an. Es bietet sich an, für größere Objekte eigene TypeScript Klassen anzulegen. Nest empfiehlt hier Klassen gegenüber Interfaces, da diese nicht zur Laufzeit entfernt werden und Nest mit Middlewares oder Pipes zusätzliche Transformationen vornehmen kann.

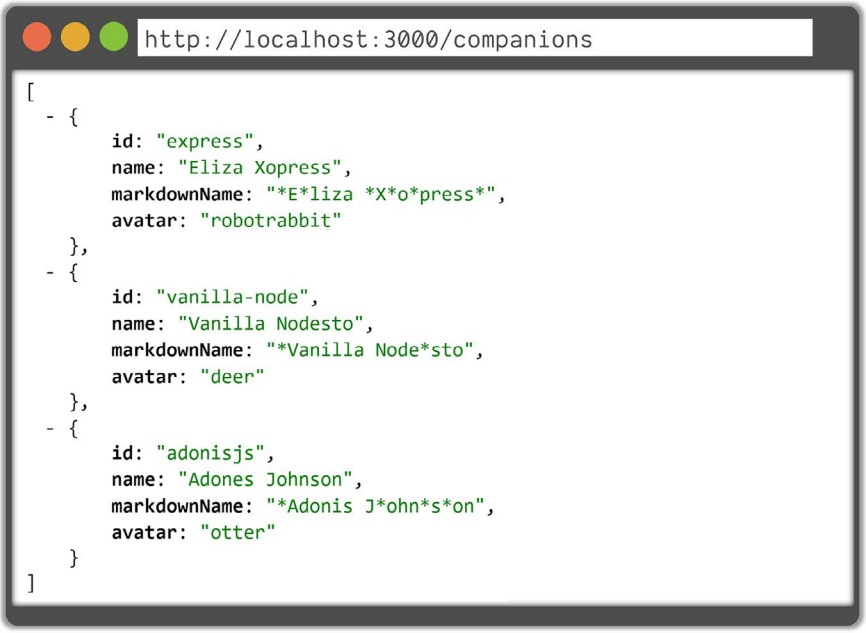
```typescript
import { Body, Controller, Post } from '@nestjs/common';

class ContactDto {
  readonly myId: string;
  readonly contactId: string;
}

@Controller('contact')
export class ContactController {

  @Post()
  sayHello(@Body() contactDto: ContactDto): string {
    const { myId, contactId } = contactDto;
    // … Freund mit ID myId sagt 'Hallo' zu Freund mit ID contactId
    return response;
  }
}
```

Zum Code →

Implementierung einer POST- Route, die über ihren Body das zuvor als TypeScript Klasse definierte Datentransfer-Objekt entgegennimmt.

Das Ergebnis

Mit dem Befehl *npm start* können wir unser Backend kompilieren und starten lassen. Über die URL *http://localhost:3000/companions* können wir uns alle *Bekannten* als JSON Objekt anfordern. Die gleiche Route würde unser Angular Client nutzen. Über die POST-Route kann unsere App gegenüber dem Backend eine Kontaktanfrage zu stellen.



Das Ergebnis der GET-Route über http://localhost:3000/companions.

Darüber hinaus

Die vorherigen Abschnitte geben einen ersten Einstieg in die Backend-Entwicklung mit Nest. Mit Hilfe der vorgestellten Konzepte lassen sich bereits erste Full-Stack Anwendungen schreiben. Wir wollen nun noch einen kurzen Blick auf weitere nützliche Bestandteile von Nest werfen.

Unit Testing

Wie wir gesehen haben, erstellt die Nest CLI automatisch *.spec*-Dateien, in denen wir Tests für unsere Controller und Services schreiben können. Dabei nutzt Nest Jest als Testing Framework, welches auch zunehmend in der Angular Community Anklang findet. Der Aufbau und die Syntax folgen dabei wie bei Angular mit Karma und Jasmine dem BDD Stil. Ein manuelles Aufsetzen einer Testinfrastruktur entfällt, und wir können zeitgleich zur Umsetzung unsere Tests schreiben.

Swagger Dokumentation

Ähnlich wie man im Frontend Tools wie Storybook nutzen kann, um Komponenten ohne eine laufende Anwendung in Isolation zu entwickeln, möchte man vielleicht auch das Backend unabhängig eines darauf zugreifenden Clients implementieren. Hierzu bietet Nest die Möglichkeit, eine interaktive API-Dokumentation mit Hilfe von Swagger zu erstellen.

Als erstes fügen wir das Swagger Plugin zu unserem Projekt hinzu:

```
> npm install --save @nestjs/swagger swagger-ui-express
```

Hinzufügen der Abhängigkeiten von Swagger und Swagger-UI.

Anschließend initialisieren wir Swagger beim Bootstrapping in der *main.ts* und können dort erste Dokumentations-Metadaten hinterlegen:

```
1  import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
2  // ...
3
4  async function bootstrap() {
5    // ...
6
7    const options = new DocumentBuilder()
8      .setTitle('ngCompanion Backend')
9      .setDescription('The ngCompanion API description.')
10     .setVersion('1.0')
11     .addTag('companions')
12     .build();
13   const document = SwaggerModule.createDocument(app, options);
14   SwaggerModule.setup('api', app, document);
15
16   // ...
17 }
18
19 bootstrap();
```

Zum Code →

In der main.ts initialisieren wir die automatische Dokumentation mit Hilfe von Swagger.

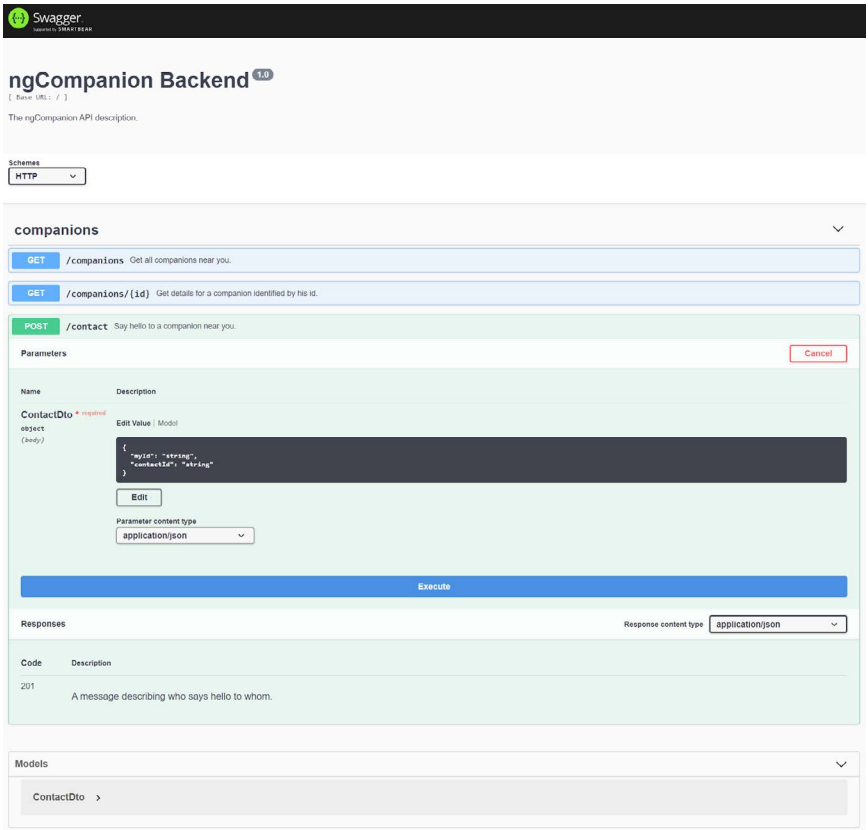
Nun haben wir über Dekoratoren die Möglichkeit, unsere API zu beschreiben. Alle möglichen Dokumentationsarten können der Nest Homepage entnommen werden. Wir können beispielsweise unsere GET-Route mit folgendem Decorator versehen:

```
import { ApiOperation, ApiResponse, ApiUseTags } from '@nestjs/swagger';
// ...

@ApiUseTags('companions')
@Controller('companions')
export class CompanionsController {
  constructor(private service: CompanionsService) {}

  @Get()
  @ApiOperation({ title: 'Get all companions near you.' })
  @ApiResponse({ status: 200, description: 'An array of companions near you.' })
  async getAll(): Promise<Companion[]> {
    return this.service.findAll();
  }

  // ...
}
```

Zum Code →

Annotation der GET-Route mit den Dekoratoren @Api... des Swagger Plugins.

Starten wir nun wieder unser Backend, können wir unter *http://localhost:3000/api* auf unsere Dokumentation zugreifen und sogar mit dem Backend interagieren. Über die Oberfläche können wir Routen-Parameter setzen oder ganze POST-Bodies definieren und brauchen so dazu kein lauffähiges Frontend.



Die Swagger Dokumentation über http://localhost:3000/api.

Eine Lösung zu jeder Aufgabe

Über diese Funktionalitäten hinaus bietet Nest eine Vielzahl verschiedener Hilfsmittel zur Lösung gängiger Problemstellungen. Deren vollständige Vorstellung würde den Rahmen des Artikels sprengen, die offizielle Dokumentation sollte aber als gute Anlaufstelle dienen. Beispielsweise gibt es auch Unterstützung für WebSockets, GraphQL, NoSQL und SQL Datenbanken, Microservices und vieles mehr.

Fazit

Als Angular Frontend-Entwickler, der gelegentlich in die Backend-Entwicklung eintauchen möchte, findet man mit NestJS ein Framework, bei dem man ähnlichen Konzepten begegnet, das sich in saubere, wegweisende Architekturen eingliedern und mit den gleichen Sprachen und Tools wie im Frontend entwickeln kann. Wie Angular besitzt auch Nest eine hervorragende Developer-Experience, einen leichten Einstieg durch seine CLI und eine ausgezeichnete Online-Dokumentation. Es liefert nicht zuletzt durch sein sehr umfangreiches Set an Plugins und Rezepten einen großen Baukasten, von dem man profitieren kann. Aus meiner Sicht sind Angular und Nest zwei Frameworks, die schlichtweg füreinander geschaffen wurden und mit deren Hilfe die Entwicklung von Full-Stack Web-Anwendungen zu einem abgerundeten, angenehmen Erlebnis wird.

Der Autor

Christian Liebel

Christian Liebel ist Consultant bei der Thinktecture AG in Karlsruhe, wo er moderne Businessanwendungen auf Basis von Angular und .NET Core umsetzt. Für seine Communityaktivitäten wurde er als Google Developer Expert (GDE) und Microsoft Most Valuable Professional (MVP) ausgezeichnet. Als Mitglied der Web Applications Working Group des W3C hält er den Blick auf kommende Webschnittstellen.

Angular-Performance: So zünden Sie den Turbo

Die Ansprüche von Anwendern an die Performance von Anwendungen sind über die letzten Jahre stetig gewachsen. Für alle Apps – ob nativ oder im Web – gilt: Nur was sich flüssig bedienen lässt, nutzen Anwender auch gerne. Dieser Artikel beschreibt die Stellschrauben in Angular, die die Laufzeitperformance Ihrer Anwendung maßgeblich beeinflussen.

Eine der zentralen Funktionen des Single-Page-Application-Frameworks Angular ist seine Unterstützung für *Data Binding*: Das Framework kümmert sich darum, dass Daten aus der Komponentenklasse an der gewünschten Stelle in der View angezeigt werden. Bei einer Änderung des Datenmodells aktualisiert das Framework automatisch die View und hält diese damit synchron. Dieser Prozess nennt sich *Change Detection*. Jeder Angular-Komponente ist ein *Change Detector* zugeordnet. Bei jeder Änderung innerhalb der Anwendung werden diese einmal durchlaufen, unidirektional von oben nach unten (siehe Abb. 1).

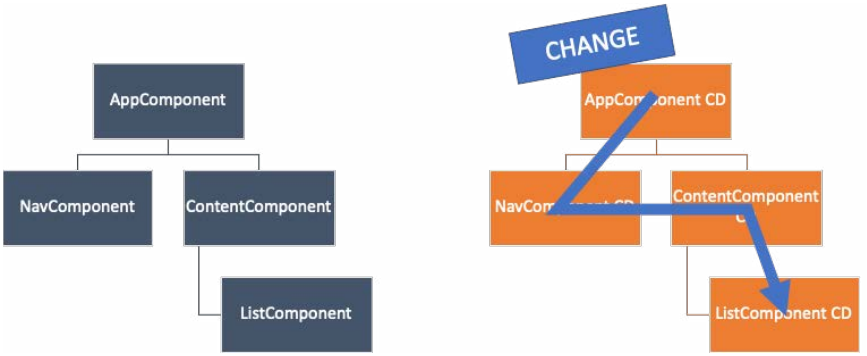


Abb. 1: Jeder Angular-Komponente ist ein Change Detector zugeordnet

Dieser Prozess läuft scheinbar magisch im Hintergrund. Wohl die wenigsten Angular-Entwickler haben irgendeine Berührung mit dem Prozess – bis sie auf Performanceprobleme stoßen. Denn die Change Detection hängt mit der Laufzeitperformanz von Angular-Apps direkt zusammen.

Um eine Änderung im Datenmodell zu erkennen, verwendet Angular eine Bibliothek namens *Zone.js*. Diese Bibliothek ist quelloffen und wird ebenfalls vom Angular-Team herausgegeben. Nach eigenen Angaben ist die Bibliothek ein *Meta-Monkey-Patch* und stellt einen *asynchronen Ausführungskontext* zur Verfügung. Unter einem Monkey Patch versteht man die Manipulation von Systemfunktionen zur Laufzeit: Zone.js überschreibt viele Browserfunktionen, durch die sich der Zustand innerhalb der Webanwendung ändern könnte, mit einer eigenen Implementierung. So wird etwa die Methode *setTimeout()*, die zur verzögerten Ausführung einer Funktion verwendet werden kann, durch eine von Zone.js bereitgestellte, kompatible Implementierung ersetzt. Die Bibliothek ruft intern die Browserimplementierung auf und erlangt Kenntnis darüber, wann eine asynchrone Operation gestartet und beendet wird. Das wiederum versteht man unter dem asynchronen Ausführungskontext. Gleiches gilt auch für die Funktion *setInterval()*, viele Browserereignisse wie click oder mousemove und weitere Browserfunktionen.

Zone.js ist ein integraler Bestandteil des Angular-Frameworks und wird daher mit jedem Projekt mitinstalliert. Noch bevor Angular initialisiert wird, startet zunächst Zone.js und erstellt den globalen Ausführungskontext (Zone). Mit dem Start von Angular wird von dieser Zone die sogenannte *NgZone* abgezweigt. Darin laufen alle Ereignisse, die durch den Quelltext innerhalb der Angular-Anwendung ausgelöst werden. Wann immer eine asynchrone Operation innerhalb der NgZone beendet wurde und keine weiteren Aufgaben mehr anstehen, löst das Framework einen Change-Detection-Zyklus aus (siehe Abb. 2).

Abb. 2: Funktionsweise der Change Detection in Angular

Das Verhalten des Frameworks ist sehr nützlich, kann in bestimmten Fällen aber auch gegen die Entwickler arbeiten: Performanceprobleme treten in Angular typischerweise beim besonders häufigen Aufruf der Change Detection in Kombination mit lang andauernden Change-Detection-Zyklen auf. Zu solchen Szenarien kommt es etwa bei Verwendung sogenannter High-Frequency-Events wie mousemove oder scroll: Registriert sich die Angular-Anwendung auf eines dieser Ereignisse, wird die Change Detection für fast jeden Pixel einer Mausbewegung oder eines Scrollvorgangs ausgelöst. Noch schlimmer verhält es sich bei der Verwendung der Methode *requestAnimationFrame()*, die für 2D- oder 3D-Visualisierungen genutzt wird und die entsprechend der Bildschirmwiederholfrequenz aufgerufen wird – auf vielen Geräten also 60 mal pro Sekunde.

Die Dauer eines Change-Detection-Zyklus wiederum hängt von der Anzahl der Data Bindings ab und wie schnell die dahinterliegenden Werte abgerufen werden können. Entwickler sollten darauf achten, nur so viele Bindings wie nötig zu verwenden. Grids sollten etwa Virtual-Scrolling-Mechanismen nutzen, um Bindings nur auf die sichtbaren

Zeilen zu beschränken. Weiterhin sollten Entwickler vermeiden, die View auf rechenintensive Getter oder Methoden zu binden, da diese für jeden Change-Detection-Zyklus aufgerufen würden. Im Idealfall sollten die Bindings direkt auf ein Feld binden, da reine Leseoperationen sehr schnell durchgeführt werden können.

Checks sparen mit OnPush

Eine weitere Möglichkeit, die Change-Detection-Dauer in der Anwendung zu verkürzen, bietet die Change-Detection-Strategie ***OnPush***. Normalerweise werden im Rahmen eines Zyklus sämtliche Komponenten geprüft. Mithilfe von OnPush kann die Ausführung der Change Detection auf die Änderung von Eingabeparametern einer Komponente eingeschränkt werden oder imperativ erfolgen. Ansonsten nimmt die Komponente nicht am Zyklus teil (siehe Abb. 3).

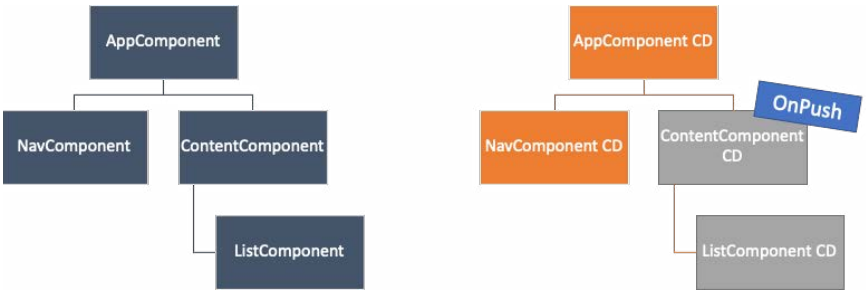


Abb. 3: Aussetzen aus dem Change-Detection-Zyklus mit OnPush

Um diese Strategie einzusetzen, wird der Komponenten-Decorator um die Eigenschaft *changeDetection* erweitert. Dieser wird der Wert *ChangeDetectionStragegy.OnPush* zugewiesen:

```
1  @Component({
2    selector: 'my-component',
3    template: '{{ foo }}',
4    changeDetection: ChangeDetectionStrategy.OnPush
5  })
6  export class MyComponent {
7    @Input()
8    foo: string;
9  }
```

Zum Code →

Die Komponente wird fortan nur noch nach templateseitigen Änderungen seiner Input-Properties Teil der Change Detection sein, in diesem Beispiel also bei Änderung des gebundenen Wertes für *foo*:

```
1  <my-component [foo]="bar"></my-component>
```

Zum Code →

Dabei gilt es zu beachten, dass Angular aus Performancegründen intern den zuvor gesetzten Wert mit dem aktuellen vergleicht *(oldValue === newValue)*. Nur wenn der Vergleich *false* zurückgibt, wird das Binding aktualisiert. Werden Objekte eingereicht, muss sich zum Auslösen der Change Detection zwingend die Referenz ändern, da der Vergleich

andernfalls *true* zurückgibt und kein Update geschieht. In diesem Fall können Immutable-Bibliotheken helfen, die für jede Änderung immer eine neue Objektinstanz zurückgeben.

Können darüber hinaus weitere Ereignisse auftreten, die zu einer Aktualisierung oder Oberfläche führen sollen (etwa ein Abruf von Daten über einen Service), müssen Entwickler die Komponente imperativ wieder zum Teil der Change Detection werden lassen. Das geht über die Schnittstelle *ChangeDetectorRef*, die sich per Dependency Injection anfordern lässt und Zugriff auf den Change Detector der jeweiligen Komponente gewährt:

```
1  data: string;
2  subscription: Subscription;
3
4  constructor(private dataService: DataService, private cdRef: ChangeDetectorRef) {}
5
6  ngOnInit() {
7    this.subscription = this.dataService.updates$.subscribe(data => {
8      this.data = data;
9      this.cdRef.markForCheck();
10   });
11 }
12
13 ngOnDestroy() {
14   this.subscription.unsubscribe();
15 }
```

Zum Code →

Dies geschieht über die Methode *markForCheck()*, die aufgerufen werden muss, sobald sich eine Änderung auf einem anderen Weg ergeben kann. Hierbei ist jedoch Vorsicht geboten: Vergessen Entwickler, die Methode aufzurufen, so kann es passieren, dass View und Model nicht mehr synchron sind.

Perfect Match: OnPush & Async-Pipe

Um dieser Unhandlichkeit zu begegnen, hat das Angular-Team die Async-Pipe eingeführt. Diese Pipe nimmt Observables oder Promises entgegen und ruft bei einer Änderung automatisch die Methode *markForCheck()* auf. Darüber hinaus kümmert sich die Pipe auch darum, sich zum Beispiel von einem Observable wieder ordnungsgemäß zu deregistrieren. Der Code von oben kann unter Verwendung der Async-Pipe folgendermaßen gekürzt werden:

```
1  data$: Observable<string>;
2
3  constructor(dataService: DataService) {
4    this.data$ = dataService.updates$;
5  }
```

Zum Code →

Im Template bindet der Entwickler die View dann lediglich auf das Observable-Feld und wendet darauf die Async-Pipe an.

```
1  {{ data$ | async }}
```

Zum Code →

Bei Verwendung von Observables oder Promises in Kombination mit der Async-Pipe kann es dann nicht mehr dazu kommen, dass View und Model auseinanderlaufen. Dank OnPush wird die Komponente nur noch geprüft, wenn sich wirklich eine Änderung ergeben hat. In Kombination mit der Async-Pipe eignet sich OnPush auch hervorragend als Standardstrategie.

Aus der Change Detection komplett aussteigen

Außerdem können Komponenten auch komplett aus der Change Detection aussteigen. In diesem Fall werden sie beim Abarbeiten des Zyklus grundsätzlich übersprungen. Dazu kann auf dem ChangeDetectorRef die Methode *detach()* aufgerufen werden. Dieses Vorgehen eignet sich vor allem dann, wenn eine Komponente temporär ausgeblendet wird, aber nicht komplett zerstört werden soll. Änderungen, die währenddessen am Datenmodell vorgenommen werden, werden in der View nicht nachgeführt. Bei Bedarf kann über die Methode *detectChanges()* eine lokale Change Detection nur für die aktuelle Komponente und ihre Kindkomponenten durchgeführt werden. Um schließlich wieder an der regulären Change Detection teilzunehmen, wird auf derselben Schnittstelle die Methode *reattach()* aufgerufen.

Dauer des Change-Detection-Zyklus messen

Um flüssig zu wirken, sollte ein einzelner Change-Detection-Zyklus deutlich kürzer sein als 16 Millisekunden, also die Dauer eines einzelnen Frames bei 60 fps. Das Angular-Team empfiehlt, den Zyklus kürzer als drei Sekunden zu halten. Um festzustellen, ob eine Anwendung in dieser Hinsicht überhaupt ein Problem aufweist, gibt Angular Entwicklern ein Werkzeug an die Hand, das die Dauer des Change-Detection-Zyklus stoppt. Das Werkzeug muss jedoch erst aktiviert werden. Dazu wird die Datei *main.ts*, die den Bootstrapping-Prozess der Angular-Anwendung definiert, folgendermaßen angepasst:

```
1  platformBrowserDynamic().bootstrapModule(AppModule)
2    .then(moduleRef => {
3      const applicationRef = moduleRef.injector.get(ApplicationRef);
4      const appComponent = applicationRef.components[0];
5      enableDebugTools(appComponent);
6    })
```

Zum Code →

Anschließend stehen die Debug-Tools zur Laufzeit der Anwendung auf der Entwicklerkonsole zur Verfügung. Die Konsole ist in den Entwicklertools des jeweiligen Webbrowsers zu finden, die in den meisten Fällen über die Tastenkombination F12 und das Anwählen der Registerkarte *Console* geöffnet werden können. Dort kann dann die Methode *ng.profiler.timeChangeDetection()* aufgerufen werden. Das Tool misst die durchschnittliche Dauer eines Change-Detection-Zyklus: Es läuft mindestens 500 Millisekunden oder fünf Zyklen lang (je nachdem, was *zuletzt* erreicht ist). Das Ergebnis wird anschließend auf der Konsole ausgegeben. Damit erhalten Entwickler ein Indiz, ob etwaige Performanceprobleme durch eine lange Zyklusdauer hervorgerufen werden könnten. Im Falle einer Blanko-Angular-Anwendung mit wenigen Bindings liegt die Dauer eines Change-Detection-Zyklus bei einem Bruchteil einer Millisekunde (siehe Abb. 4).

Abb. 4: Ausführung des Angular-Debug-Tools

Nachdem nun Techniken gezeigt wurden, die die Dauer des Change-Detection-Zyklus reduzieren können, werden als nächstes Methoden vorgestellt, um die Anzahl dieser Zyklen zu reduzieren.

Change-Detection-Zyklen zusammenlegen

Angular 9 bringt einen neuen Modus mit sich, der in speziellen Fällen dazu beitragen kann, mehrere Change-Detection-Zyklen zu einem zusammenzulegen. Im folgenden Beispiel löst sowohl ein Klick auf die Schaltfläche als auch einer auf das umschließende Element ein Ereignis aus. Normalerweise würde es hier zu zwei Change-Detection-Zyklen kommen.

```
1  <div (click)="foo()">
2    <button (click)="bar()">Test</button>
3  </div>
```

Zum Code →

In den meisten Fällen dürfte es hier jedoch genügen, einen einzigen Zyklus nach dem Ablauf beider Ereignishandler auszuführen. Dafür kann seit Angular 9 beim Starten der Anwendung in der Datei *main.ts* im Konfigurationsobjekt der Methode *bootstrapModule()* die Eigenschaft *ngZoneEventCoalescing* mit dem Wert *true* hinterlegt werden.

```
1  platformBrowserDynamic()
2    .bootstrapModule(AppModule, { ngZoneEventCoalescing: true })
3    .catch(err => console.error(err));
```

Zum Code →

Opt-out: Zone temporär deaktivieren

Wenn Entwickler die Zone temporär umgehen möchten, können sie auf die Schnittstelle NgZone zurückgreifen. Diese lässt sich über die Dependency Injection des Frameworks anfordern. Auf der Schnittstelle finden sich unter anderem die Methoden *runOutsideAngular()* und *run()*. Beide Methoden nehmen eine Funktion entgegen, die dann außerhalb respektive innerhalb der NgZone ausgeführt wird.

```
1  constructor(ngZone: NgZone) {
2    ngZone.runOutsideAngular(() => {
3      // runs outside Angular zone, for performance-critical code
4
5      ngZone.run(() => {
6        // runs inside Angular zone, for updating view afterwards
7      });
8    });
9  }
```

Zum Code →

Performancekritischer Code sollte außerhalb der Angular-Zone ausgeführt werden. So führt etwa der Aufruf von *requestAnimationFrame()* nicht mehr zu einem Change-Detection-Zyklus (siehe Abb. 5).

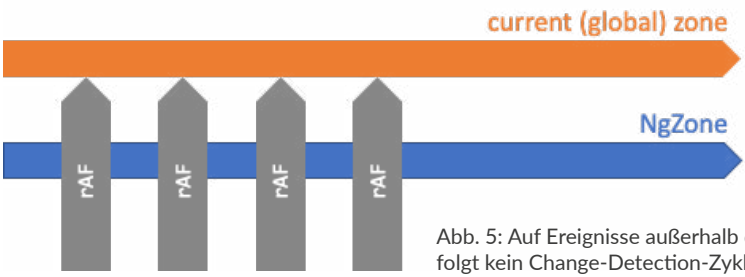


Abb. 5: Auf Ereignisse außerhalb der NgZone folgt kein Change-Detection-Zyklus

Änderungen, die außerhalb der NgZone durchgeführt werden, werden umgekehrt allerdings nicht mehr von Angular erkannt. Es kann also dazu kommen, dass die View vom tatsächlichen Stand im Datenmodell abweicht. Muss die View basierend auf einem außerhalb der Zone ermittelten Ergebnis aktualisiert werden, so kann über die Methode *run()* die NgZone auch wieder betreten und Aktualisierungen durchgeführt werden.

Zyklen sparen: Zone-Patches deaktivieren

Entwickler haben jedoch nicht immer die Möglichkeit, über die gezeigte Methode aus der NgZone auszusteigen. Insbesondere, wenn eine 2D- oder 3D-Visualisierung durch eine Drittanbieterbibliothek gesteuert wird, kann beispielsweise der Aufruf von *requestAnimationFrame()* nicht mit einem *runOutsideAngular()* umschlossen werden. In diesen Fällen haben Entwickler jedoch die Möglichkeit, Zone-Patches gezielt abzuschalten. Dies erfolgt in der Datei polyfills.ts, die noch vor der Ausführung von Angular geladen wird. Hier wird auch die Bibliothek Zone.js importiert. Bevor dies geschieht, sind folgende Zeilen zu finden, die standardmäßig auskommentiert sind:

```
(window as any).__Zone_disable_requestAnimationFrame = true;
// disable patch requestAnimationFrame

(window as any).__Zone_disable_on_property = true;
// disable patch onProperty such as onclick

(window as any).__zone_symbol__UNPATCHED_EVENTS = ['scroll', 'mousemove'];
// disable patch specified eventNames
```

Zum Code →

Die erste Zeile deaktiviert den Zone-Patch für die schon mehrfach genannte Methode *requestAnimationFrame()*. Das eignet sich für den Fall, wenn die Methode von einer Drittanbieterbibliothek aufgerufen wird und die Aufrufe keinen Change-Detection-Zyklus nach sich ziehen sollen. Die zweite Zeile deaktiviert die Patches von DOM-Ereignishandlern wie onclick. In der dritten Zeile haben Entwickler schließlich die Möglichkeit, bestimmte Ereignisse zu ignorieren. Hier sind bereits die beiden kritischen Ereignisse scroll und mousemove eingetragen. Die angegebenen Ereignisse erreichen die NgZone also nicht und führen nicht mehr zu einem Change-Detection-Zyklus. Doch auch in diesen drei Fällen ist wieder Vorsicht geboten: Wird ein Patch deaktiviert, ist er nun für die komplette Anwendung abgeschaltet. Auch Angular-Ereignishandler, die auf ein mousemove reagieren, führen bei Verwendung der dritten Zeile nicht mehr zu einem Change-Detection-Zyklus. In einem solchen Handler vorgenommene Änderungen würden in der View also nicht nachgeführt.

Zone komplett abschalten

Darüber hinaus ist es auch möglich, Zone.js komplett abzuschalten. Das bedeutet im Umkehrschluss, dass Entwickler bei jeglichen Änderungen selbst die Change Detection auslösen müssen. In Angular geht das mithilfe der Schnittstelle ApplicationRef, die sich über die Dependency Injection des Frameworks anfordern lässt:

```
1  constructor(applicationRef: ApplicationRef) {
2    applicationRef.tick();
3  }
```

Zum Code →

Um die Zone für die Anwendung zu deaktivieren, wird in der Datei main.ts im Konfigurationsobjekt für die Methode *bootstrapModule()* der Eigenschaft ngZone der Wert *noop* zugewiesen. Die Zone-Implementierung wird dann gegen einen Dummy ausgetauscht, der keine Change Detection auslöst.

```
1  platformBrowserDynamic().bootstrapModule(AppModule, {
2    ngZone: 'noop'
3  })
```

Zum Code →

In durch Angular hervorgerufene Performanceprobleme kann eine derart konfigurierte Anwendung praktisch nicht laufen, umgekehrt ergibt sich jetzt aber die Gefahr, dass Entwickler das Aufrufen der Change Detection vergessen könnten und die View damit eventuell die Synchronisation mit dem Datenmodell verliert. Für Anwendungen mit höchsten Ansprüchen an die Performance könnte dies aber eine interessante Option sein.

Zusammenfassung: So bleiben Angular-Apps fast and fluid
Entwickler sollten darauf achten, den Change-Detection-Zyklus so kurz wie möglich zu halten. Dies gelingt durch das Reduzieren der Bindinganzahl auf das erforderliche Minimum, das Vermeiden von Data Bindings auf rechenintensive Getter oder Funktionen, die Verwendung der Change-Detection-Strategie OnPush oder der ChangeDetectorRef-Schnittstellen. Darüber hinaus sollten so wenige Zyklen wie nötig ausgelöst werden, durch das temporäre oder vollständige Deaktivieren der Zone beziehungsweise ihrer Patches.

Angular-Entwickler sollten diese Mechanismen unbedingt beherrschen, da bei falscher Handhabung View und Model auseinanderlaufen könnten. Wer diese Grundregeln beachtet, dürfte mit seiner App in keine Performanceprobleme laufen.

Der Autor

David Müllerchen

David Müllerchen (auch als webdave bekannt) ist ein Trainer, Berater und Google Developer Expert für Webtechnologien und Angular. Seine Leidenschaft ist Wissen zu teilen, in Vorträgen und Workshops, in-house oder in der Öffentlichkeit, auf Meetups und Konferenzen. Außerdem ist David im Orga-Team des Angular Hamburg Meetups und der Angular Konferenz NG-DE.

Angular-Testing mit Cypress.io

Normalerweise ist mein Thema ja eher Angular und das komplette Ökosystem dazu. Aber auch Testing ist mir eine Herzensangelegenheit. Ich möchte in diesem Artikel eine Möglichkeit präsentieren, wie man Webanwendungen testen kann.

Wir schauen uns in diesem Zusammenhang das Tool Cypress an. Wir lernen, wie man Cypress in den Angular-Workspace integriert und es nutzt. Mein Ziel ist es, möglichst viele Entwickler für Tests zu begeistern.

Beim Thema Testen muss man zwischen zwei Arten unterscheiden.

- Unit Tests (Code based)
- E2E Tests (User flow)

Was sind Unit Tests?

Unit Tests sind dem einen oder anderen eventuell auch als Module Tests bekannt. Hierbei werden einzelne Teile einer Anwendung (Units) getestet. So ist es zum Beispiel möglich, einen Service zu testen, ohne dass die komplette Anwendung (oder gar das Backend) laufen muss. Dadurch ist etwas Wunderbares möglich: Test Driven Development (TDD).

Ich schreibe also einen Test für meinen Service, wie er am Ende funktionieren muss. Dann lasse ich den Test laufen und implementiere solange, bis der Test grün ist. Dadurch reduziere ich die Gefahr, zu viel Logik zu implementieren. Hierfür kann man verschiedene Tools nutzen, z. B.: Mocha, Jasmine und Jest. Gerade in Agilen Teams sind Unit Tests verbreitet.

Vorteile:

- Fehler in der Implementierung werden schnell entdeckt.
- Ein Test kann auch als Dokumentation dienen.

Nachteile:

- Kann sehr zeitraubend sein, gerade wenn das erwartete Verhalten erst (wasserfallartig) zur Entwicklungszeit definiert wird.
- Bei Refactoring müssen meist alle Tests neu geschrieben werden.

Was sind E2E Tests?

E2E ist die Kurzform für End-to-End Tests. Sie werden im Gegensatz zu Unit Tests auf die komplette Applikation angewendet. Wie der Begriff End-To-End vermuten lässt, können alle Layer in den Test einbezogen werden. Vom User Interface (UI) bis zum Backend. Ich kann zum Beispiel testen, ob Interaktionen in der UI die erwarteten Aktionen im Backend ausführen (CRUD von Daten).

Wir reden hier von Tests, die ganze User-Stories abdecken. Diese Tests werden meist am Ende der Entwicklung (eines Features) geschrieben. Wenn ich ein Refactoring der Applikation vorhabe, sollte ich eine möglichst hohe Testabdeckung haben, damit ich sicherstellen kann, dass das Refactoring nicht den Flow der Applikation zerstört hat. Hierfür kann man verschiedene Tools nutzen, z. B.: Cypress, Protractor, Nightwatch.js, und Puppeteer.

Vorteile:

- Ich kann sicherstellen, dass meine Applikation wie erwartet funktioniert (durch alle Layer hindurch).
- Es werden 'echte' User-Interaktionen getestet.

Nachteile:

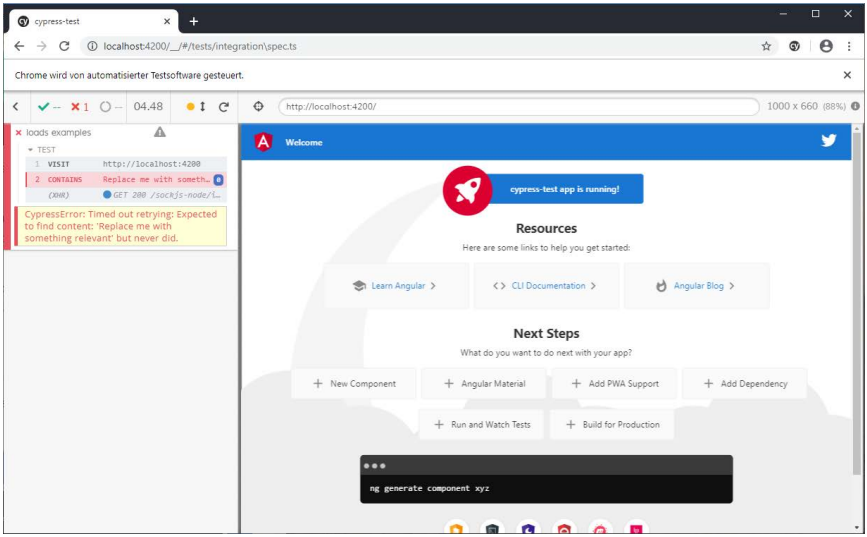
- Diese Tests sind sehr fragil, wenn das UI sich ändert.
- Es besteht die Gefahr, dass diese Tests (da sie am Ende eines Entwicklungszyklus geschrieben werden) beim Projektverantwortlichen den Anschein erwecken, man könne sie doch auch leicht durch menschliche Tester ersetzen (irgendwann testet der das dann eventuell auch).

Sie sehen, Unit Tests und E2E Tests haben ihre eigenen Daseinsberechtigungen und sind am besten in Kombination anzuwenden.

In diesem Artikel möchte ich über E2E Testing schreiben. Ich bin vor einiger Zeit auf Cypress aufmerksam geworden und finde es super.

Was ist Cypress?

Cypress ist ein E2E Testing Tool, das mit einer eigenen Electron App kommt, in der die zu testende App ausgeführt wird.



Der Vorteil daran: Cypress verhält sich wie ein Interceptor oder Proxy. Alle Kommunikation, die von der zu testenden App ausgeht, kann von Cypress überwacht, manipuliert und gemockt werden.

Was alle E2E Tools gemeinsam haben, ist, dass sie nicht für ein bestimmtes JavaScript Framework gebaut wurden. Man kann damit jede Web App testen.

Wie kann ich Cypress für meine Angular App nutzen?

Ich starte hier mit einem frisch begonnenen Angular Projekt (*ng new cypress-test*).

Cypress ist Framework-unabhängig, d.h. es können alle Webseite damit getestet werden, es muss keine Angular App sein.

Mit Angular hat man den großen Vorteil, dass man dank der CLI die Umstellung von Protractor (als Standard E2E Testing Tool) zu Cypress vollautomatisch durchführen kann. Voraussetzung ist ein Projekt, welches mit der CLI Version 6+ gebaut wurde.

Alles, was wir dafür tun müssen, ist der CLI zu sagen, sie solle bitte Cypress installieren:

```
ng add @briebug/cypress-schematic
```

Dies veranlasst die CLI dazu, Cypress zu installieren (*npm install cypress*) und das Projekt für Cypress zu konfigurieren.

Die CLI fragt dann, ob Protractor entfernt werden soll.

Would you like to remove Protractor from the project?

Klar, es macht für uns keinen Sinn, zwei E2E Tools in einem Projekt zu haben.

Daraufhin werden alle Projector Dateien gelöscht und Cypress Dateien angelegt.

```
DELETE e2e
```

Als nächsten kommt dann die automatische Konfiguration der CLI, um Cypress zu nutzen.

```
CREATE cypress.json (48 bytes)
CREATE cypress/tsconfig.json (196 bytes)
CREATE cypress/integration/spec.ts (123 bytes)
CREATE cypress/support/commands.ts (838 bytes)
CREATE cypress/support/index.ts (689 bytes)
UPDATE package.json (1401 bytes)
UPDATE angular.json (4333 bytes)
```

Was bedeutet das?

In der *angular.json* ist die Task e2e konfiguriert. Diese wird durch die *@briebug/cypress-schematic* umgeschrieben, um die Cypress-Funktionalität zu nutzen.

```
        "e2e": {
          "builder": "@angular-devkit/build-angular:protractor",
          "options": {
            "protractorConfig": "e2e/protractor.conf.js",
            "devServerTarget": "my-project:serve"
          },
          "configurations": {
            "production": {
              "devServerTarget": "my-project:serve:production"
            }
          }
        }


         "e2e": {
          "builder": "@briebug/cypress-schematic:cypress",
          "options": {
            "devServerTarget": "cypress-test:serve",
            "watch": true,
            "headless": false
          },
          "configurations": {
            "production": {
              "devServerTarget": "cypress-test:serve:production"
            }
          }
        },
```

Zum Code →

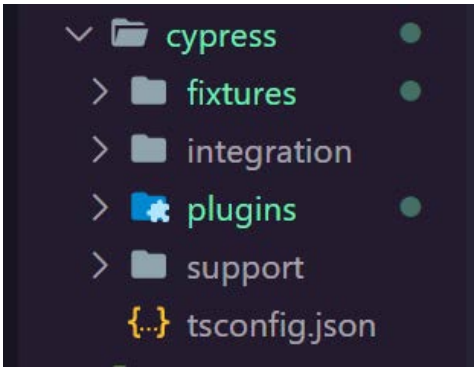
Das war es auch schon mit der Vorbereitung. Es ist nur ein Befehl und die Bestätigung, dass man Protractor ersetzen möchte.

Der erste Test

Ausgeführt werden die Tests wie gewohnt (wenn man vorher schon mit Protractor gearbeitet hat).

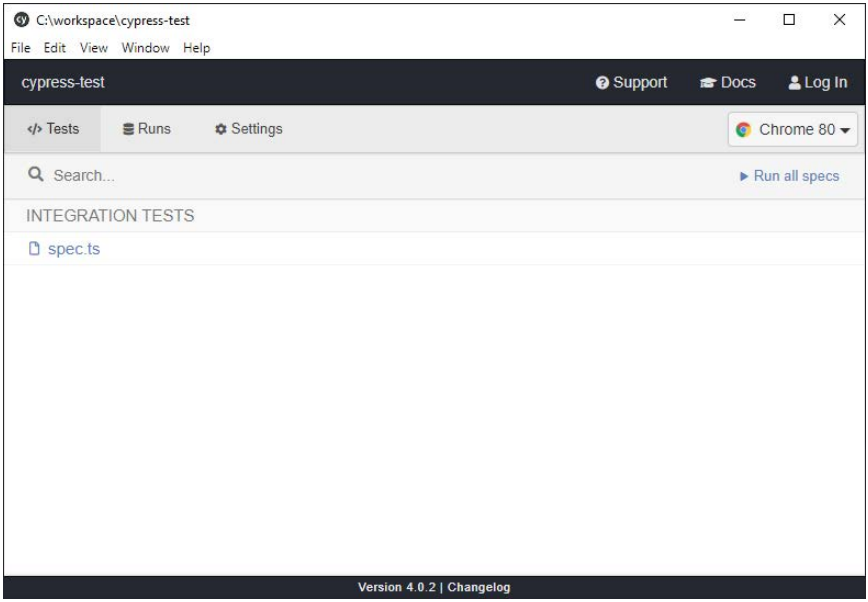
```
ng e2e
```

Beim ersten Mal werden eine Ordnerstruktur und einige Config-Dateien angelegt.



1. *fixtures:* Hier können Mockdaten abgelegt werden, die dann im Test genutzt werden können.
2. *integration:* Hier werden die Tests geschrieben.
3. *plugins:* Hier können Plugins (wenn man diese benötigt) eingebunden werden.
4. *support:* Hier können eigene Commands geschrieben oder existierende überschrieben werden.

Danach wird das Cypress-UI gestartet.



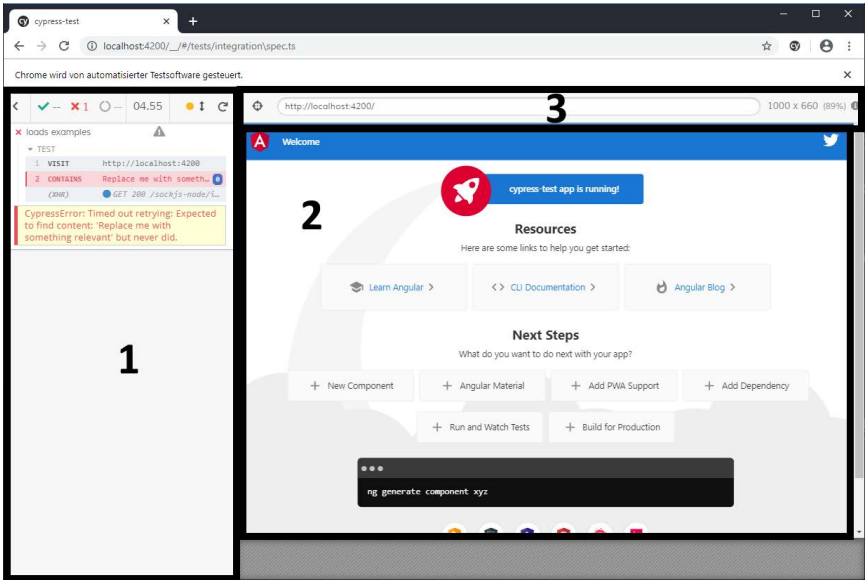
Über den Button *Run all specs* kann man alle vorhandenen Testdateien und die darin geschriebenen Tests ausführen. In der Liste *INTEGRATION TESTS* sind alle Testdateien aufgeführt, und man hätte hier die Möglichkeit, einzelne Testdateien auszuführen.

Wir haben bisher nur eine Datei, aber man kann gut erahnen, wie einfach man seine Tests organisieren kann: Einfach in unterschiedliche Dateien auslagern.

Der Cypress Client

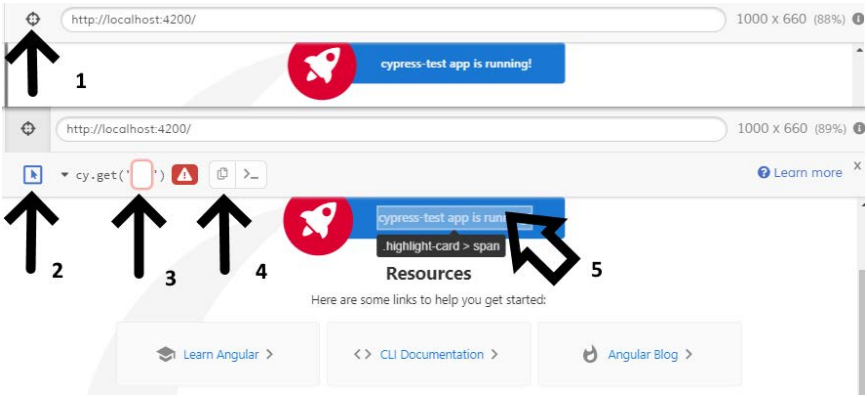
Wir klicken auf *Run all specs* und der Cypress Client wird gestartet. Der Client ist eine Electron App, die unsere Website lädt und die Tests ausführt.

Den Client kann man in drei Bereiche teilen. Bereich 1 ist der Reporter-Bereich, hier werden die Testergebnisse ausgegeben. Wenn man mit der Maus über die durchlaufenen Tests fährt, sieht man im Bereich 2 die Interaktionen, die in der App ausgeführt wurden. Im Bereich 3 findet man die Adressleiste und den Playground.



Klickt man auf die Playground-Schaltfläche – das Fadenkreuz (1), wird der Playground eingeblendet. Hier hat man als Werkzeug den Selectorknopf (2). Einmal angeklickt, kann man nun in der App ein Element auswählen und bekommt einen Selektor (3) für dieses Element vorgeschlagen und kann den kompletten Aufruf direkt über die Copy-Schaltfläche (4) in den Zwischenspeicher legen.

Hat man den Selectorknopf geklickt, bekommt man beim Überfahren von Elementen in der App auch Hinweise zu den Elementen (5).



Initial sind noch keine sinnigen Tests vorhanden, nur ein „must Failing" Test, da der erwartete Text *'Replace me with something relevant'* nicht gefunden wird.

```
1  it("loads examples", () => {
2    cy.visit("http://localhost:4200");
3    cy.contains("Replace me with something relevant");
4  });
```

Zum Code →

Aber wir können hier schon etwas Wichtiges sehen.

Alle Tests müssen in einem *it()* geschachtelt sein. Das *it()* hat keine echte Bedeutung, es geht hier darum, Tests zu schreiben, die für den Entwickler lesbar sind.

Wir können die Tests auch noch weiter organisieren. Zum Beispiel können wir Test Suits definieren, um darin dann die Tests zu schreiben.

Eine Test Suit wird über *describe()* definiert:

```
describe('Next Steps', () => {...})
```

Zum Code →

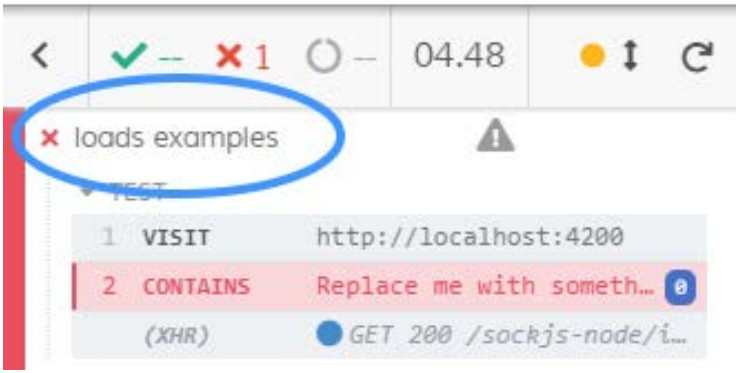
Innerhalb eines *describe* kann ich beliebig viele *it()* oder auch weitere *describe* definieren, um meine Tests zu organisieren. Außerdem gibt mir *describe* die Möglichkeit, mit Preflies oder Rollbacks zu arbeiten.

Hier ein Beispiel:

```

describe("Hooks", () => {
  before(() => {
    /* Wird einmalig VOR allen Tests ausgeführt */
  });

  after(() => {
    /* Wird einmalig NACH allen Tests ausgeführt */
  });

  beforeEach(() => {
    /* Wird VOR JEDEM Test ausgeführt */
  });

  afterEach(() => {
    /* Wird NACH JEDEM Test ausgeführt */
  });
});
```

Zum Code →

Das *it()* wie auch das *describe()* haben zwei Parameter. Der erste ist ein String und wird im Testergebnis als Titel ausgegeben. So ist das Testergebnis nachher gut lesbar und aussagekräftig.



Der zweite Parameter ist dann unser Test.

Zur Erinnerung, E2E Tests sind wie ein User Test. Alles was der Nutzer tut, kann ich auch im Test tun.

Was möchte ich testen? Was sollte ich testen?

Testen sollte man:

- Alles, was in den User Stories steht. Die Expectations sind geradezu perfekt als Test Titel (erster Parameter eines *it*) geeignet.
- Alles, was einem kritisch erscheint. Alles, was beim Entwickeln etwas mehr Hirnschmalz erfordert hat, oder was einem unheimlich vorkam.
- Alles, was schon einmal kaputt war. Es gibt nichts Ärgerlicheres, als einen Fehler später nochmal zu finden.

Was ich hier testen möchte:

- Beim Besuchen der Seite soll der Nutzer mit dem Namen des Projekts begrüßt werden, in dieser Art: ***ProjektName app is running***.
- Beim Klick auf eine der Next Steps soll in der Anzeige (sieht aus wie eine Console) eine bestimmte Ausgabe angezeigt werden.

CY Methoden

In den Tests hat man Zugriff auf das globale Cypress Objekt (***cy***). Über ***cy*** habe ich Zugriff auf viele Methoden, wir schauen uns einige davon an. Die erste Methode ist mit die wichtigste.

Das globale Cypress Objekt gibt Zugriff auf Methoden, die ich zum Testen nutzen kann:

- ***.visit(url):*** Damit kann ich im Browser (im Cypress Client) eine bestimmte URL aufrufen. In unserem Beispiel wollen wir die Standard URL unseres Angular Projektes aufrufen. ***cy.visit(„http://localhost:4200“);*** Der User besucht unsere Website.
- ***.get(selector):*** Hiermit kann ich Elemente auswählen, um Aktien oder Prüfungen darauf auszuführen. Der Selektor ist dabei der Standard CSS Selektor.
- ***.contains(content):*** Damit kann geprüft werden, ob ein Text/Content gefunden wurde. Es kann hiermit aber auch gefiltert werden. (Das zeige ich gleich noch.)
- ***.should(chainer, value):*** Mit dieser Methode können Prüfungen durchgeführt werden. Der Chainer ist hierbei die Erwartung (Expectation), die gegen den Wert (Value) geprüft wird. (Auch das sehen wir gleich).

Unser erster wirklicher Test:
In der *app.component.html* finden wir Folgendes.

```
1
2       <span>{{ title }} app is running!</span>
3
```

Zum Code →

Wir wollen prüfen, ob *cypress-test app is running!* in einem Span ausgegeben wird.

```
1  it(`Beim besuche der Seite soll der Nutzer mit dem Namen das Projekts begrüsst werden, in form von: ProjektName app is running` () => {
2    cy.visit('http://localhost:4200');
3    cy.get('span').contains('cypress-test app is running!');
4  });
```

Zum Code →

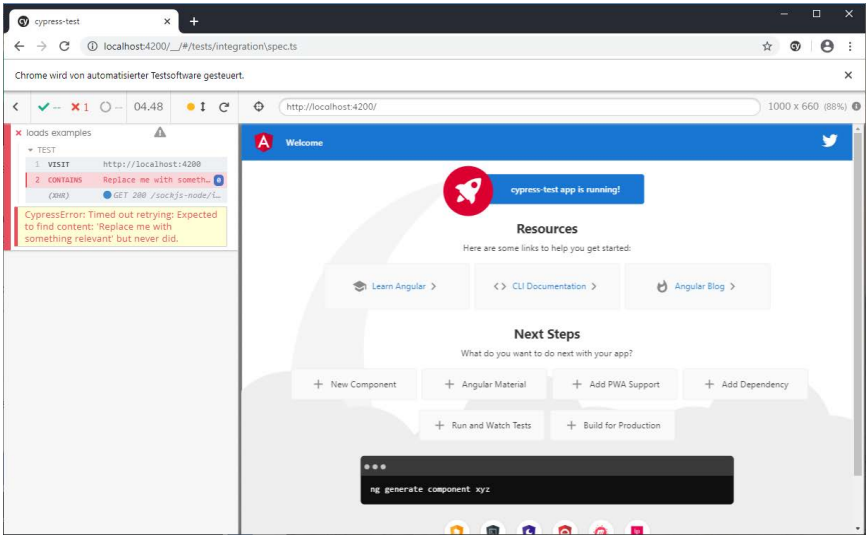
Okay, was passiert hier?

1. Wir haben einen Test Case a definiert mit dem Titel unseres ersten Tests.
2. Wir rufen die Startseite der App auf.
3. Wir suchen nach einem Span, welches unseren erwarteten Text beinhaltet.

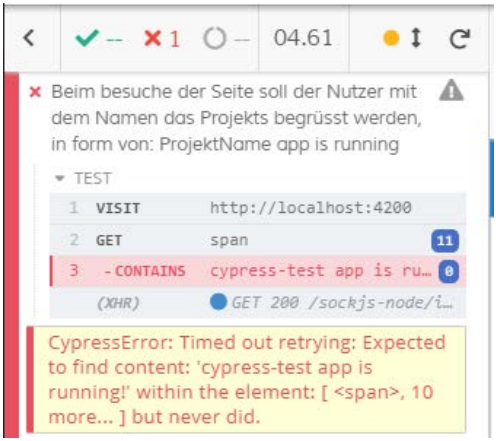
So einfach kann es sein.

Sollte es mal nicht so einfach sein, Elemente zu finden, gibt uns der Cypress Client ja den Selectorknopf als Hilfsmittel.

Wir sehen im Reporter-Bereich, dass der Test grün ist. Wenn man nun mit der Maus über die Zeile 2 des Tests fährt, sieht man in der App alle Spans, die gefunden wurden.



Würde es keinen Span mit dem gesuchten Text geben, gäbe es einen Timeout und einen Fehler.



Interagieren mit Elementen

Jedes Element, welches über *.get()* gefunden wurde, kann Aktionen empfangen. Dabei handelt es sich um die üblichen Aktionen, die ein Nutzer ausführen kann (z. B. klicken oder Text eingeben).

Hier ist eine Liste von Aktionen:

- *.click()*
- *.dblclick()*
- *.type()*
- *.clear()*
- *.check()*
- *.uncheck()*
- *.select()*
- *.trigger()*

Wir wollen nun also den zweiten Test schreiben.

Beim Klick auf eine der Next Steps soll in der Anzeige (sieht aus wie eine Konsole) eine bestimmte Ausgabe angezeigt werden.

Also legen wir erstmal ein neues *it()* mit dem Titel an:

```
1  it(`Beim Klick auf eine der Next Steps soll in der Anzeige eine Bestimmte Ausgabe angezeigt werden.`, () => {
2    cy.visit('http://localhost:4200');
3  });
```

Zum Code →

Nun wollen wir alle „Next Steps“-Schaltflächen testen.

Ich habe mir ein kleines Objekt für alle „Next Steps" und die dazugehörigen Meldungen gebaut.

```
const commands = {
  'New Component': 'ng generate component xyz',
  'Angular Material': 'ng add @angular/material',
  'Add PWA support': 'ng add @angular/pwa',
  'Add Dependency': 'ng add _____',
  'Run and Watch Tests': 'ng test',
  'Build for Production': 'ng build --prod'
};
```

Zum Code →

Ich möchte jetzt über dieses Objekt iterieren und in jeder Iteration einen Test ausführen.

```
it(`Beim Klick auf eine der Next Steps soll in der Anzeige eine Bestimmte Ausgabe angezeigt werden.`, () => {
  cy.visit('http://localhost:4200');
  for (const step in commands) {
    cy.get('span')
      .contains(step)
      .click();
  }
});
```

Zum Code →

Damit habe ich schon mal sichergestellt, dass die Schaltflächen existieren. Nun muss ich noch prüfen, ob der erwartete Text angezeigt wird.

Dafür müssen wir uns das Element holen und auf den Inhalt prüfen.

```
1   cy.get('div.terminal').should('contain', commands[step]);
```

Zum Code →

Damit hätten wir auch diesen Test durchgeführt.

Hier mal das komplette *it()*:

```
1  it(`Beim Klick auf eine der Next Steps soll in der Anzeige eine Bestimmte Ausgabe angezeigt werden.`, () => {
2    cy.visit('http://localhost:4200');
3    // tslint:disable-next-line:forin
4    for (const step in commands) {
5      cy.get('span')
6        .contains(step)
7        .click();
8      cy.get('div.terminal').should('contain', commands[step]);
9    }
10 });
```

Zum Code →

Alias

Mit Cypress haben wir die Möglichkeit, Elemente als Alias abzulegen, um auf diese mehrfach zugreifen zu können. Mit der methode *.as('myAlias')* legen wir den Alias an und können auf diesen dann über die Methode *.get('@myAlias')* zugreifen. Das vorangestellte @ gibt Cypress hier den entscheidenden Hinweis, dass es sich hierbei um einen Alias handelt.

Das würde man in einem *beforeEach()* tun.

```
describe('Cypress Test', () => {
  beforeEach(() => {
    cy.visit('http://localhost:4200');
    cy.get('span').as('spans');
    cy.get('div.terminal').as('terminal');
  });
  ...
});
```

Zum Code →

Diesen Alias können wir nun in unserem zuletzt geschriebenen Test nutzen.

```
      cy.get('@spans')
        .contains(step)
        .click();
      cy.get('@terminal').should('contain', commands[step]);
```

Zum Code →

Unsere App ist ja sehr limitiert, aber in einem echten Projekt hätte man Formulare, die man ausfüllen und abschicken würde.

Ein Test könnte so aussehen:

```
1    it('create User', () => {
2      cy.get('button.new-user').click();
3
4      cy.get('.headline').contains('Nutzer anlegen');
5
6      cy.get('input[formcontrolname="name"]').type('Hannes');
7
8      cy.get('button.save').should('be.enabled').click();
9    });
```

Zum Code →

HTTP-Aufrufe

Wir können auch prüfen, ob das Absenden der Daten funktioniert. Dafür müssen wir das Feature aktivieren, das in dem Servermodul enthalten ist.

Dieses Modul bietet die Möglichkeit, alle HTTP-Kommunikation zu überwachen, zu manipulieren oder zu beantworten.

Das bedeutet, wir brauchen kein Backend ansprechen.

Zuerst muss das Servermodul gestartet werden, dies passiert auch im *beforEach()*.

```
1  cy.server();
```

Zum Code →

Ich habe noch die Möglichkeit, diverse Konfigurationen an den Server zu übergeben. Diese Information finden Sie in der Dokumentation.

Wenn wir nun einen Endpunkt „wegmocken" wollen, ist das sehr einfach. Dafür wird eine Route definiert:

```
1       cy.route(url)
2       cy.route(url, response)
3       cy.route(method, url)
4       cy.route(method, url, response)
```

Zum Code →

Wir sehen, dass diese API sehr variabel ist.

Nachstehend ein Beispiel:

```
1       cy.route('http://my.api.com', { name: 'Hannes' });
```

Zum Code →

Solange die Response so schön klein ist, ist es auch okay (Achtung, persönliche Meinung!) diese direkt im Mock zu schreiben. Aber das entspricht ja sehr selten dem echten Entwicklerleben. Dafür bietet Cypress die Möglichkeit, gemockte Daten in Form von Dateien als Response zu nutzen.

Dafür ist der Ordner *cypress\fixtures* gedacht. Hier kann ich Dateien ablegen, die ich dann als Response nutzen kann. Hier liegt eine

example.json. Wenn wir diese als Response nutzen wollen, müssen wir im Mock statt dem Objekt *{ name: ‚Hannes' }* eine (seltsam aussehende) Cypress Syntax schreiben. Nämlich das Keyword *fixture:* gefolgt vom Namen der Datei.

```
1       cy.route('http://my.api.com', 'fixture:example.json');
```

Zum Code →

Natürlich ist jedem sofort aufgefallen, dass ich vom Absenden der Daten gesprochen habe. Wenn ich keine HTTP-Methode an *.route()* übergebe, ist es per Default ein GET. Ein POST Mock würde so aussehen:

```
1       cy.route('POST', 'http://my.api.com/users', '123456789').as(
2         'new-user'
3       );
```

Zum Code →

Alle POSTs auf den User Endpoint sind über den Alias *@new-user* erreichbar und können wie folgt getestet werden.

```
1         cy.get('@new-user')
2           .its('request.body')
3           .should('deep.equal', {
4             name: 'Hannes',
5             kundennummer: '1597',
6           });
```

Zum Code →

Fazit

Sie sehen, Cypress bietet eine Menge Möglichkeiten und macht dabei auch noch Spaß. Wir haben gelernt, wie man Elemente findet und welche Prüfungen wir darauf ausführen können. Wir haben die verschiedenen Methoden kennengelernt, wie wir mit DOM Elementen interagieren können. Und wir habe einen Überblick über die Möglichkeiten von Cypress bekommen.

Ich sehe anhand der Teilnehmer meiner Workshops, dass der Einstieg in Cypress sehr einfach ist und man nach kurzer Zeit eine gute Testabdeckung erreicht.

Host Europe

Sie haben weitere Fragen?

Unser Sales Team ist gerne für Sie da.

0800 626 4624

www.hosteurope.de