



ng1 zu ng2+ migrieren

Philipp Burgmer
w11k / theCodeCampus

Philipp Burgmer

<> burgmer@w11k.de | [@philippburgmer](https://twitter.com/philippburgmer)

Software-Entwickler, Trainer

Web-Technologien, Sicherheit

TypeScript, Angular </>

W11K GmbH - The Web Engineers

- <> Gegründet 2000
- Auftrags-Entwicklung / Consulting
- Web / Java
- Esslingen / Siegburg </>

theCodeCampus.de - Weiter.Entwickeln.

- <> Schulungen seit 2007
- theCodeCampus seit 2013
- Projekt-Kickoffs & Unterstützung im Projekt </>

- <> Historie & Motivation
- <> Strategien
- <> Schritt für Schritt

- <> AngularJS = Angular 1 = ng1
- <> Erstes Release: 2009 (IE8)
 - Mehr für Designer als für Entwickler
- <> Probleme
 - Performance bei großen Anwendungen
 - Keine Struktur auf Code Ebene
 - Schlechte Steuerung des Datenflusses

<> Angular = Angular 2+ = ng2 = ng

- Kein Angular 3
- Aktuell: Angular 4.1

<> Release 2.0 im September 2016

<> Moderne Konzepte umsetzen, aktuelle Standards nutzen

- Natives Modul System
- Komponentenorientiertes Design
- Vereinheitlichtes Templating
- Deutliche Steigerung der Performance
- Verbessertes Tooling

<> **Danach ist alles besser**

- Sauberer Code
- Super Performance
- Zukunftssicher
- Nie wieder `$scope.$apply()`

<> **Während dessen ist aber auch manches schlimmer**

<> Grundlegendes gleich

- SPA
- Data-Binding
- Dependency-Injection

<> AngularJS → Angular 2+

- Service & Factory → Service
- Filters → Pipes
- Directive - Template → Directive
- Directive + Template → Components
- Controllers → Components
- Module → NgModule
- Scope →

Strategien

- <> Anwendung neu schreiben
- <> Fachliche Details bekannt
 - Umsetzung deutlich schneller als Erstentwicklung
- <> Angular 2+ Wissen benötigt
 - Keine Zeit für seichten Einstieg

- <> Danach kein Legacy-Code mehr
- <> Chance technische Schulden abzubauen
- <> Kein zusätzlicher Aufwand für Schrittweise Migration
 - Upgrade & Downgrade
 - Build
- <> Danach hoch optimierte Anwendung

- <> Oft zu aufwändig / zu teuer
- <> Weiterentwicklung stockt
- <> Parallele Weiterentwicklung muss nachgezogen werden

- <> Anwendung bleibt rein AngularJS
- <> Konzepte und Best-Practises aufgreifen
 - Neuer Code mit neuen Konzeteten
 - Alter Code schrittweise angepasst
 - z.B. Smart-vs-Dump-Components
- <> Little Bang am Ende
 - Umstellung von AngularJS auf Angular

- <> Schrittweise neue Konzepte lernen & einbauen
- <> Weiterentwicklung parallel möglich
- <> AngularJS kann auch sehr komfortabel sein
 - TypeScript
 - ES Module
 - Komponenten

<> Tooling / Build

<> Kein echtes Angular 2+

- Performance
- Features (z.B. Style-Encapsulation)

<> Little Bang: sämtlichen Code anfassen

- <> Anwendung bleibt AngularJS
 - Angular 2+ wird zusätzlich eingebunden
- <> Neuer Code wird für Angular geschrieben
- <> Alter Code kann Schrittweise migriert werden
- <> Komponenten und Services können gegenseitig verwendet werden (Upgrade und Downgrade)

- <> Schrittweise neue Konzepte lernen & einbauen
- <> Weiterentwicklung profitiert sofort von Angular 2+
- <> Alten Code Schritt für Schritt migrieren
- <> Tiny Pop: Leichter Wechsel zu Angular am Ende

- <> **Angular wird als Bibliothek eingebunden**
 - Recht schwergewichtig (ohne Optimierungen des CLI)
 - Für Mobile eher ungeeignet

- <> **Tooling / Build**

- <> **Verstehen von Upgrade und Downgrade**

- <> **Nicht alles lässt sich up- bzw. downgraden**
 - Pipes bzw. Filter
 - Nicht-Komponenten-Direktiven

Die Wahrheit
liegt irgendwo dazwischen

- <> Erst Adaptieren
- <> Dann Hybrid
- <> Kurz vor Ende vielleicht doch Rest-Big-Bang

Schritt für Schritt

zur Angular Anwendung

Schritt 1

AngularJS aktualisieren

<> Für Adaptieren

- Neue Features verwenden können
- Angular 2+ Konzepte aufgreifen können

<> Für Hybrid

- `UpgradeAdapter` setzt AngularJS 1.5.3 voraus

<> Bugfixes

<> Performance

- <> Animationen
- <> Feature-Module
- <> ngTouch

- <> Kein IE8 Support
- <> strict-DI
- <> ngModelOptions
- <> ngModel Validator Pipeline
- <> ngMessages
- <> One-Time-Bindings

- <> Keine neuen Features
- <> Viele Detail-Verbesserungen
 - Teilweise Breaking-Changes
 - Animationen, \$http, jQuery, Cookies, ngMessages
- <> Performance: Expression-Parser, Compiling, Watching

<> Multi-Slot-Transclusion

<> Annäherung an Angular 2

- `module.component` als Alternative zu `module.directive`
- Lifecycle-Hooks für Direktiven (ab 1.5.3)
`$onInit()`, `$onChanges()`, `$onDestroy()` und `$postLink()`

- <> `ngModelOptions` können vererbt werden
- <> Nicht standardisierte Promise-Methoden `success` und `error` an Rückgabe von `$http` entfernt
- <> Breaking Change für Komponenten-Bindings:
keine Zuweisungen mehr vor Controller-Instanziierung

Schritt 2

TypeScript einführen

<> Warum?

- Typisierung
- Code-Strukturierung
- ES6+ Features

<> Wie?

- *.js zu *.ts umbenennen
- Tooling einbauen (z.B. WebPack)

- <> TypeScript + Module-Loader
- <> Wichtiges Feature für große Anwendungen
- <> Ein Einstiegspunkt, danach nur über import-Statements
 - Pro Software-Komponente eine Datei (ein Modul)
 - Erleichtert Navigation im Code
- <> Bibliotheken müssen per NPM installiert werden

- <> **Module-System von AngularJS ist Schrott**
 - Man kann aber nicht ohne
- <> **Trotzdem: Feature-Module**
 - Erleichtert Wechsel zu Angular 2+ Modulen später
 - Ordner-Struktur = Feature-Struktur
- <> **Angular-Module exportieren (eigene Datei)**
- <> **Angular-Module importieren und über Name einbinden**
 - AppModule bindet Feature-Module ein

- <> @types stellt Typisierung für JS Bibliotheken bereit
 - AngularJS in JS entwickelt, keine statische Typisierung
 - Community stellt Declaration-Files bereit
 - Declaration-Files beinhalten nur Typ-Informationen
- <> Abhängigkeiten in `package.json`
 - `angular` als `dependency`
 - `@types/angular` als `devDependency`

Schritt 3

An Style-Guide anpassen

<> Versuchen sich an Style-Guide und Best Practices zu halten

<> [John Papa's Style-Guide](#)

- Feature-Module und Ordner
- Direktiven mit isolated Scope
- Kein Zugriff auf fremde Daten (Controller-As)
- Routing
- ...

<> Viele Angular 2+ Konzept und Best Practices bauen darauf auf

<> Service & Controller als Klassen

- Service mit `module.service` registrieren
- Controller wie gewohnt

<> ES6+ Features nutzen

- `Promise` statt `$q`
- `for..of` statt `angular.forEach`
- `let` Variablen

Schritt 4

Angular 2+ einbinden

<> AngularJS Bootstrapping

- Automatisch über `ng-app` Attribute im Markup
- Manuell über `angular.bootstrap` Aufruf im Code

<> Bei Verwendung von `UpgradeAdapter` nur manuelles Bootstrapping möglich

- Angular 2+ Module anlegen
- Angular 2+ bootstrappen (zumindest fast)
- AngularJS bootstrappen

<> Kompliziert, aber ein mal Copy & Paste

```
1 @NgModule({
2   imports: [BrowserModule, UpgradeModule]
3 })
4 export class AppModule {
5   ngDoBootstrap() {}
6 }
7
8 platformBrowserDynamic().bootstrapModule(AppModule).then(platformRef => {
9   const upgrade = platformRef.injector.get(UpgradeModule) as UpgradeModule;
10  upgrade.bootstrap(document.body, ['app'], {strictDi: true});
11 });
```

Schritt 5

Service migrieren

- <> Service schon als Klasse implementiert
 - Kleine Anpassungen für DI
- <> Registrierung ändern: Angular 2+ UND AngularJS Module
- <> Downgrade wenn in AngularJS Code verwendet
- <> Abhängigkeiten upgraden oder ersetzen

```
1 import {downgradeInjectable} from '@angular/upgrade/static';
2
3 angular.module('app')
4   .factory('MyService', downgradeInjectable(MyService));
5
6 @NgModule({
7   providers: [
8     MyService,
9     otherServiceProvider
10  ]
11 })
12 export class AppModule {}
```

```
1 import {OpaqueToken} from "@angular/core";
2 export const OtherService = new OpaqueToken("OtherService");
3
4 export function otherServiceFactory(i: any) {
5   return i.get('otherService');
6 }
7 export const otherServiceProvider = {
8   provide: OtherService,
9   useFactory: otherServiceFactory,
10  deps: ['$injector']
11 };
12
13 export class MyService {
14   constructor(@Inject(OtherService) private other) {}
15 }
```

Schritt 6

Komponenten migrieren

- <> Controller schon als Klasse implementiert
 - Kleine Anpassungen für DI
 - Wird zu Komponenten-Klasse
- <> Template muss Angular 2+ Expression Language verwenden
- <> Registrierung ändern: Angular 2+ UND AngularJS Module
 - Downgrade wenn in AngularJS Code verwendet
 - `declaration` und `entryComponent` an Angular 2+ Module
- <> Verwendete Komponenten upgraden oder ersetzen

```
1 import {downgradeComponent} from "@angular/upgrade/static";
2
3 @Component({
4   selector: 'app-list',
5   template: '<div></div>'
6 })
7 export class ListComponent {}
8
9 angular.module('app')
10  .directive('list', downgradeComponent({
11    component: ListComponent
12  }) as angular.IDirectiveFactory);
```

Schritt 7

AngularJS entfernen

- <> Alle Komponenten auf Angular 2+ umgestellt
- <> Routen sind nur noch sehr dünne Wrapper
 - Verweisen auf Komponente
- <> Einfach zu migrieren
 - UI-Router zu Angular Router

- <> Nur noch Angular 2+ starten
- <> Alle Import-Statements zu AngularJS entfernen
- <> EntryComponents kontrollieren

- <> Tooling auf Angular CLI umstellen
 - Von hoch spezialisiertem Build profitieren
 - AOT Compile
 - Test Ausführung
- <> Neues Projekt mit CLI anlegen
- <> Projekte zusammenführen

- <> Migration mittlerweile sehr gut dokumentiert
 - [angular.io / Upgrading from AngularJS](#) mit [PhoneCat Tutorial](#)
 - [NG-Conf 2017 Vortrag](#) von Asim Hussain mit [Contacts App](#)
- <> [UpgradeAdapter](#) erspart Boilerplate
 - Nicht gescheit dokumentiert
- <> Upgrade Pfad klar ersichtlich
 - Code nicht immer ganz einfach
- <> Nicht immer Upgrade zwingend
 - TypeScript, Module, RxJS
 - gut ohne Angular 2+ möglich

Philipp Burgmer
burgmer@w11k.de
Twitter: @philippburgmer
GitHub: pburgmer

www.thecodecampus.de
@theCodeCampus