Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Niklas Kopp

Erweiterbare Architektur von Webclients in Angular zum Umsetzen mehrerer Anwendungen mit geteilter Codebasis

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Niklas Kopp

Erweiterbare Architektur von Webclients in Angular zum Umsetzen mehrerer Anwendungen mit geteilter Codebasis

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 27. Juli 2018

Niklas Kopp

Thema der Arbeit

Erweiterbare Architektur von Webclients in Angular zum Umsetzen mehrerer Anwendungen mit geteilter Codebasis

Stichworte

Angular, Software, Architektur, Web, Single Page Application, Single Page Anwendung

Kurzzusammenfassung

Dieser Bachelorarbeit untersucht, welche Lösungen das Angular Framework bietet um Single Page Applications mit einer erweiterbaren geteilten Codebasis zu entwickeln. Dies geschieht anhand einiger prototypischer Beispiele und einer realen Anwendung. Durch die modulare Struktur des Angular Frameworks bieten sich hier eine Vielzahl eleganter Möglichkeiten um gängige Muster anzuwenden und umzusetzen, von denen einige in dieser Arbeit dargestellt werden.

Niklas Kopp

Title of the paper

Extensible architecture of webclients with angular for creating multiple applications with a shared code base

Keywords

angular, software, architecture, web, single page application

Abstract

This bachelor thesis examines the solutions provided by the angular framework for developing single page applications with an extensible shared code base. It does so using some prototypic examples and a real world application. Due to the modular structure of the angular framework, there are numerous elegant possibilities to apply and implement common patterns, some of which are are discussed in this thesis.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Fragestellung . . . 1
- 1.2 Ziel der Arbeit . . . 2
- 1.3 Struktur der Arbeit . . . 2

2 Grundlagen **4**
- 2.1 Single Page Applications . . . 4
  - 2.1.1 Nachteile Klassische Webanwendungen . . . 4
  - 2.1.2 Konzept Single Page Anwendungen . . . 5
- 2.2 Grundlagen und Konzepte in Angular . . . 6
  - 2.2.1 Typescript und Angular . . . 7
  - 2.2.2 Komponenten . . . 8
  - 2.2.3 Dependency Injection und Services . . . 9
  - 2.2.4 Module . . . 11
  - 2.2.5 RxJs Integration . . . 13
  - 2.2.6 Weitere Konzepte . . . 14

3 Anwendung der Konzepte **15**
- 3.1 Multi-App Setup . . . 15
  - 3.1.1 Multi-App Setup bis Angular Version 5 . . . 16
  - 3.1.2 Multi-App Setup ab Angular Version 6 . . . 22
- 3.2 Basisservices . . . 28
  - 3.2.1 Grundlagen Basisservices . . . 28
  - 3.2.2 Beispiel Basisservice Struktur . . . 29
- 3.3 Providen von speziellen Serviceausprägungen . . . 31
  - 3.3.1 Grundprinzip Serviceausprägungen . . . 32
  - 3.3.2 Beispiel Serviceausprägungen . . . 32
- 3.4 Transclusion . . . 34
  - 3.4.1 Grundprinzip Transclustion . . . 34
  - 3.4.2 Einfaches Transclusion Beispiel . . . 34
  - 3.4.3 Beispiel Grundlayout . . . 35
  - 3.4.4 Geschachtelte Transclusion . . . 38
- 3.5 Extension-Points . . . 40
  - 3.5.1 Grundprinzip Extension-Points . . . 40
  - 3.5.2 Beispiel Extension-Points . . . 41
  - 3.5.3 Extension via Provide-Multi . . . 45

3.6 Fallbeispiel Rails 48
3.6.1 Anwendungsarchitektur 49
3.6.2 Verwendete Konzepte 49

4 Zusammenfassung und Ausblick **53**
4.1 Zusammenfassung 53
4.2 Ausblick 54

Listings

3.1 .angular-cli.json 18
3.2 package.json 19
3.3 .angular-cli.json 20
3.4 .angular-cli.json 21
3.5 public_api.ts 26
3.6 tsconfig.json 27
3.7 entity-layer-service-base.ts 30
3.8 concrete-entity-layer-service.ts 31
3.9 layout-no-transclusion.html 35
3.10 base-layout.component.html 37
3.11 layout-transclusion.html 37
3.12 layout-multi-slot-transclusion.html 38
3.13 base-layout.component.html 38
3.14 base-right.component.html 39
3.15 menu-button-plugin.ts 41
3.16 extension-point.component.html 42
3.17 map-extension.ts 46
3.18 map-service.ts 47

1 Einleitung

Das Angular Framework erfreut sich immer größerer Beliebtheit 16. Viele moderne Webanwendungen werden als Single Page Applications realisiert und können als Rich-Client auch clientseitig eine hohe Komplexität aufweisen. In dieser Arbeit sollen (bezogen auf einen speziellen Anwendungsbereich) einige Möglichkeiten erkundet werden, wie diese Komplexität mit den Werkzeugen von Angular durch gute Strukturen und Architektur beherrschbarer werden kann.

1.1 Fragestellung

Die Arbeit beschäftigt sich mit der Frage, wie es mit dem Angular Framework 1 technisch möglich ist, erweiterbare Strukturen zu schaffen um mehrere Anwendungen auf einer gemeinsamen Basis aufzusetzen.

Diese Frage kommt z.B. beim Entwicklung einer Software als Produkt und nicht als eigenständige Lösung auf. Hierbei ist das Produkt eine Basiskomponente, die für jeden Kunden unverändert bleibt. Ausgehend von dieser Basis kann dann eine auf den Kunden und seine spezielle Fachlichkeit zugeschnittene Lösung entwickelt werden. Je mehr Funktionalität (und auch Fachlichkeit) sich von den unterschiedlichen Fachlichkeiten abstrahieren und in die Basiskomponente auslagern lässt, desto geringer der Entwicklungsaufwand bei der Entwicklung einer Lösung für einen neuen Kunden mit anderen Anforderungen.

Dies setzt voraus, dass es genug Gemeinsamkeiten zwischen den Anforderungen der jeweiligen Kunden gibt und somit der Aufwand einer geteilten Codebasis gerechtfertigt ist. Zudem ist es wichtig, dass die Basiskomponente eine gute Erweiterbarkeit gewährleistet. Es muss Möglichkeiten geben, die vorhandenen Klassen und Komponenten so zu erweitern, anzupassen oder zu konfigurieren, dass die speziellen Anforderungen mit möglichst wenig Overhead implementiert werden können. Die Änderung sollten nach dem “Open-Closed Principle” möglich sein, ohne die Basis hierfür modifizieren zu müssen (vlg. Starke, 2015, S. 71-73).

Fast alle der behandelten Konzepte lassen sich auch für die Entwicklung einer einzelnen Anwendung nutzen, um diese besser zu strukturieren und etwa Gemeinsamkeiten mehrerer Module in ein Basismodul zu extrahieren.

Die Frage nach den technischen Möglichkeiten von Angular zur Umsetzung einer solchen Architektur soll in dieser Arbeit anhand einiger Beispiele beantwortet werden.

Es soll im Rahmen der Arbeit nicht darum gehen, wie fachliche Schnitte gesetzt werden können um die gemeinsame Fachlichkeit der Kunden ebenfalls in die Basiskomponente zu verlagern, sondern lediglich um die technischen Möglichkeiten und eine Beispielhafte Umsetzung einzelner nötiger Schritte für eine solche Architektur.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, an einem überschaubaren Beispiel verschiedene Techniken zu demonstrieren. Die entstehende Software soll übersichtlich bleiben und die Komplexität einer wirklichen Anwendung nur exemplarisch andeuten. Trotzdem sollen die Beispiele auf komplexe Anwendungen übertragbar sein.

Als Beispiel dient eine prototypische georeferenzierte Anwendung, die verschiedene Layer auf einer Karte darstellt. Hierbei werden kleinere Features auf verschiedene Art umgesetzt, um als Beispiel für die möglichen Vorgehensweisen zu dienen, wie sie in dieser Arbeit beschrieben werden. Die Beispiele sollen hierbei als Proof-of-Concept dienen und kein konkretes Problem lösen. Anhand von Codebeispielen und Beschreibungen können die Vorgehensweisen auf konkrete Probleme in anderen Kontexten übertragen werden. Zudem soll als Fallbeispiel eine Anwendung dienen, die nicht im Rahmen dieser Arbeit, sondern in einem realen Kundenprojekt entwickelt wird. Um die Komplexität der Beispiele zu reduzieren, basieren die meisten jedoch auf der prototypischen Anwendung, die speziell zu diesem Zweck entwickelt wurde.

1.3 Struktur der Arbeit

Im Grundlagenteil (2) werden zunächst das Konzept von Single Page Applications und die Grundlagen von Angular erläutert. Hierbei werden die im weiteren Verlauf wichtigen Konzepte anhand einiger kleiner Beispiele dargestellt um das Verständnis des Anwendungsteils zu vereinfachen.

Im Anwendungsteil (3) wird anhand praktischer Codebeispiele zunächst der Setup von Multi-App Projekten mit dem Angular Command Line Interface (CLI) und dann eine Reihe von Konzepten zum Erstellen einer erweiterbaren geteilten Codebasis erläutert. Diese Konzepte erheben keinen Anspruch auf Vollständigkeit sondern sind eine Reihe von Möglichkeiten, die sich in der praktischen Arbeit als nützlich erwiesen haben. Zuletzt wird an dem Beispiel der Anwendung Rails dargestellt, wie sich einige der behandelten Konzepte auch in einer produktiv entwickelten Anwendung umsetze ließen.

2 Grundlagen

In diesem Kapitel soll zunächst auf Single Page Applications und die Gründe eingegangen werden, diese klassischen Webanwendungen vorzuziehen sofern eine möglichst interaktive Anwendung entwickelt werden soll. Dann soll ein Überblick über das Angular-Framework zum Erstellen solcher Single Page Applications und die zugrundeliegenden Konzepte geboten werden. Hierbei geht es nicht darum, das Framework in aller Tiefe zu erklären, sondern lediglich darum die im Verlauf dieser Arbeit wichtigsten Konzepte und Prinzipien vorzustellen.

2.1 Single Page Applications

Da Angular speziell für Single Page Applications ausgelegt ist, soll kurz darauf eingegangen werden was genau eine Single Page Application ausmacht. Hierfür sollen zunächst die Nachteile klassischer Webanwendungen dargestellt werden, um die Vorteile von Single Page Applications besser nachvollziehbar zu machen. Dies ist ausführlicher in Kuuskeri (2011) beschrieben und soll in diesem Abschnitt kurz erläutert werden.

2.1.1 Nachteile Klassische Webanwendungen

Klassische Webanwendungen haben einige entscheidende Nachteile. Hier werden die Seiten meist Serverseitig gerendert, Aktionen des Clients sorgen für ein Laden einer weiteren Seite, in die dann das Ergebnis der Aktion hineingerendert wird.

Da die Seiten serverseitig gerendert werden, für bestimmte Anwendungsfälle (Animationen, schnelle Reaktion auf Klicks) aber nach wie vor Clientseitige Veränderungen des DOM nötig sind, wird der View Teil der Anwendung auf Server und Client verteilt und hat keinen festen Platz. Ein nötiges Neuladen der Seite bei jeder größeren Aktion des Clients sorgt zudem für unangenehme Verzögerungen und stellt so einen großen Nachteil von klassischen Webanwendungen gegenüber nativer Desktopanwendungen dar.

Bei jedem Neuladen der Seite wird der JavaScript Teil der Anwendung neu gestartet, da der Browser die Seite inklusive JavaScript vollständig neu aufbaut. Der Zustand kann also nicht in der Client-Anwendung gehalten werden, sondern muss über Cookies und/oder Sessions auf dem Server gehalten werden. Diese unscharfe Verteilung der Zuständigkeiten und starke Kopplung zwischen Client und Server sorgen für schwer überschau- und wartbare Anwendungen.

2.1.2 Konzept Single Page Anwendungen

Bei Single Page Applications wird eine viel stärkere Trennung von Server und Client angestrebt. Der Server bietet Services per ReST an. Diese Services sind, wie bei ReST üblich, zustandslos. Der Server trifft keine Annahmen über den Client der einen Request schickt, jeder Request wird für sich und unabhängig von vorigen Requests behandelt. Zustand hält der Server nur für die von ihm angeboten und persistenten Resourcen. Der Server ist so nicht auf einen speziellen Client beschränkt sondern bietet seine Dienste beliebigen Clients (oder auch weiteren Serverdiensten) an. So lässt sich der Server völlig isoliert testen und betreiben. Clients können ohne Änderungen am Server hinzugefügt oder ausgetauscht werden und mit etwas Vorsicht bei Änderungen der API können Server und Client auch in voneinander entkoppelten Zyklen entwickelt werden.

Der Client-Teil der Anwendung funktioniert wie eine klassische Desktopanwendung, die im Browser läuft. Die Anwendung verändert den HTML-DOM nach belieben, um die GUI darzustellen. All dies geschieht ohne das Laden von HTML-Seiten vom Server, oder das Neuladen der Seite, der Client muss jederzeit selbstständig in der Lage sein, die GUI zu rendern. Services des Servers können per ReST in Anspruch genommen werden, dies kann (und sollte) asynchron im Hintergrund passieren.

In vielen Fällen muss der Nutzer so nicht aktiv auf einen Request warten, da der Client benötigte Resourcen schon im voraus laden kann, ohne dass der Nutzer dies merkt. Zudem kann der Nutzer (anders als beim Neuladen der Seite) die Anwendung wie gewohnt weiter bedienen während ein Request lädt. Ein Loading-Indicator kann den offenen Request visualisieren, das Ergebnis wird dann bei erfolgreichem Laden in die GUI gerendert. Der ReST der GUI bleibt bedienbar und kann sogar parallel weitere Requests an den Server anstoßen. Dies wäre mit dem Klassischen Modell bei Webanwendungen nicht möglich.

Sieht sich der Nutzer zum Beispiel eine lange Liste von Beiträgen an, kann das Laden der nächsten Beiträge angestoßen werden, sobald der Nutzer sich dem Ende der Seite nähert. Wenn der Nutzer das Ende der Seite dann tatsächlich erreicht hat, sind die nächsten Beiträge bereits im Hintergrund geladen und auf die Seite gerendert, der Nutzer spürt somit keinerlei Wartezeit. Dieses Vorgehen bietet viele Verbesserungsmöglichkeiten der Reaktionszeit, da zukünftige Nutzeraktionen antizipiert und benötigte Server-Resourcen im voraus geladen werden können, um die Latenz bei Nutzeraktionen zu reduzieren.

Da die Anwendung im Client nicht bei jedem Request neugestartet wird, wie beim Neuladen der Seite erforderlich, kann der Client einen eigenen Zustand halten. Dieser kann eingetragene Werte in Formularen, ausgewählte Werkzeuge, oder den Fortschritt im Bestellvorgang beinhalten. Der Client befreit den Server so davon, Zustand für jeden einzelnen Client halten zu müssen. Das sorgt zum einen für eine sauberere und Zustandslose Schnittstelle, zum anderen eröffnet dies die Möglichkeit der horizontalen Skalierung der Serverkomponenten, da für den Client nicht wichtig ist, ob ein Request vom selben Server bearbeitet wird, wie der vorherige.

Wenn Persistenter Zustand benötigt wird (z.B. Nutzerspezifische Konfiguration) kann dieser als Resource auf dem Server gespeichert und beim Neustart der Anwendung per ReST geladen werden.

Bei allen Vorteilen die eine solche Single Page Anwendung bietet, stellt das dynamische Rendern der GUI auf einer einzelnen HTML-Seite eine große Herausforderung dar, die ohne entsprechendes Framework extrem mühsam umzusetzen wäre.

2.2 Grundlagen und Konzepte in Angular

In dieser Arbeit soll im speziellen Angular und dessen Möglichkeiten und Lösungen genauer untersucht werden. Das Framework hat sich in den letzten Jahren (laut Stackoverflow Entwicklerumfage) neben Node.js zum meistgenutzten Framework entwickelt 16. Mit der Unterstützung von Google ist die Wahrscheinlichkeit hoch, das Angular auch in Zukunft weiterentwickelt wird, was das Risiko reduziert eine Anwendungen auf einem Framework aufzusetzen, das in wenigen Jahren komplett obsolet ist.

Obwohl es weitere Frameworks für Single Page Applications gibt, die ähnlich populär sind (React z.B.), konzentriert sich diese Arbeit auf das Angular Framework. Ein Vergleich mit anderen Frontend-Bibliotheken ist nicht Teil der Arbeit.

Mit Angular steht ein Open Source Framework zur Verfügung, das sehr gut an die Bedürfnisse von Entwicklern komplexerer Single Page Applications zugeschnitten ist. Dependency Injection, Annotationsbasierte Konfiguration, ein Komponenten- und Modulsystem, sowie gute Unterstützung für Unit und Integrationstests sorgen dafür, dass gegenüber der Backend Entwicklung mit einem Framework wie Spring 15 nur wenige Abstriche gemacht werden müssen. Da Angular in erster Linie für Single Page Applications entwickelt wurde, ist es möglich Anwendungen zu entwickeln, die sich wie Rich-Clients bedienen lassen, trotzdem aber die Vorteile von Web-Clients (keine Installation, Automatische Updates, Plattformunabhängigkeit) mitbringen.

Im folgenden Abschnitt sollen diese Aspekte etwas detaillierter dargestellt werden.

2.2.1 Typescript und Angular

Eine Besonderheit von Angular ist, dass das Framework in Typescript geschrieben ist und verwendet werden sollte. Typescript ist eine von Microsoft entwickelte Programmiersprache, die auf JavaScript basiert. Die Sprache ist ein Superset von JavaScript, das heißt jedes JavaScript Programm ist automatisch auch ein gültiges Typescript Programm.

Typescript bringt im Gegensatz zu JavaScript, wie der Name schon vermuten lässt, eine statische Typisierung mit. Um Interoperabilität mit ungetyptem JavaScript Code zu ermöglichen sind diese Typisierungen optional. Für viele JavaScript Bibliotheken existieren Typdefinitionen für Typescript, die die möglichen Typen der JavaScript APIs einschränken und so besser mit Typescript Code benutzbar machen.

Da die Browser kein Typescript unterstützen, wird der Quellcode vor der Ausführung im Browser zu JavaScript (ES5) compiliert, was von der Angular Tool-Chain übernommen wird. So steht dem Entwickler eine Sprache mit starkem Typsystem wie aus Java oder C# zur Verfügung, die zudem alle aktuellen Sprachfeatures von ECMAScript 2017 (ES8) einschließt, ohne auf Kompatibilität mit Browsern Rücksicht nehmen zu müssen. Die starke Typisierung ermöglicht das Entdecken von Fehlern zur Compilezeit, die bessere Code Analyse durch Tools und somit viele aus dem Backend bekannte Komfortfunktionen wie Autovervollständigung, navigierbare Methoden und Klassen und Refactoring-Unterstützung (vgl. Malcher u. a., 2017, S. 28-29, S. 43).

Zudem weisen empirische Studien (Fischer und Hanenberg, 2015, S. 165), (Petersen u. a., 2014, S. 221), (Hanenberg, 2010, S. 303) darauf hin, dass das Entwickeln in einer typisierten Sprache produktiver möglich ist als mit dynamisch typisierten

Sprachen wie JavaScript, gerade wenn es darum geht fremden Code zu verstehen. Interessanterweise ergaben die genannten Studien, dass ein reines Autocomplete-Feature für die schwach typisierten Sprachen nur einen kleinen Vorteil bot, während der Vorteil von typisierten Sprachen deutlich schwerer ins Gewicht fiel.

Diese Entscheidung auf Typescript zu setzen, macht es Backend Entwicklern deutlich einfacher, die gelernten Konzepte aus dem Backend auch im Frontend umzusetzen. Viele der Grundkonzepte in Angular lassen sich sehr gut nutzen, um Anwendungen erweiterbar zu strukturieren. Im Folgenden soll kurz auf diese Konzepte eingegangen werden, da sie als Grundlage für die spezielleren Schritte des Designs einer geteilten Basiskomponente dienen werden.

2.2.2 Komponenten

Komponenten sind in Angular die Verbindung mit der HTML-Oberfläche. Über HTML-Templates und Event- bzw. Data-Binding stellen sie eine Verbindung zwischen dem Programmcode und dem DOM im Browser her (vgl. Malcher u. a., 2017, S. 64-73). Dies sorgt für eine gute technische Schichtung, da die GUI deklarativ über ein Template beschrieben und direkt an das Datenmodell gebunden wird. Eine Änderung am Modell wird direkt in der GUI reflektiert, ohne dass hierzu weiterer Code nötig wäre.

Diese von Angular bereitgestellte Implementierung des Model-View-Controller-Musters (vgl. Starke, 2015, S.124) ermöglicht die direkte Interaktion des fachlichen Codes mit dem Model, ohne sich um die Konsistenz der View kümmern zu müssen. Dies sorgt nach dem Prinzip der Separation of Concerns dafür, dass der Code der für das Binden und Aktualisieren der HTML-Seite zuständig ist, nicht mit dem Code der Anwendung vermischt wird, da er im Angular Framework getrennt vom Anwendungscode verortet ist.

Kommunikation zwischen Komponenten kann unter anderem über das Template erfolgen. Hierzu werden Properties und Events mit @Input()- und @Output()-Annotationen versehen und können so die Konfiguration und Kommunikation mit der Komponente über das HTML-Template abwickeln. Die @Input()-Annotation wird an Properties bzw. Property-Setter gesetzt und ermöglichen so das Setzen eines Wertes aus dem HTML-Template heraus. Dies kann hilfreich sein um den Zustand aus der Komponente per Button oder anderem Input zu steuern. Die @Output()-Annotation kann an ein Observable bzw. einen EventEmitter gebunden werden (mehr zu Observables in Abschnitt 2.2.5). So kann die Komponente eventbasiert in die

andere Richtung kommunizieren und Nachrichten an übergeordnete Komponenten schicken.

```
...
export class SubComponent {
        @Input()
        public inputProperty: number;
        @Output()
        public outputEvent: EventEmitter<number> = new EventEmitter();
        ...
}
```

```
        <app-sub-component
            [inputProperty]="42"
            (outputEvent)="onSubEventFired($event)">
        </app-sub-component>
```

In diesem Beispiel ist die Syntax im HTML-Template zu sehen. Eckige Klammern stehen für das Binden vom HTML-Template an die Komponente über die @Input()-Annotationen, runde Klammern für das Binden in der Rückrichtung über die @Output()-Annotation. Dies ermöglicht die Kommunikation der Komponenten deklarativ über das Template und bietet zumindest für kleine Kommunikationsschnittstellen eine gute Möglichkeit, die direkt sichtbar und klar definiert ist.

Wenn sehr viel Kommunikation zwischen den Komponenten stattfindet, oder die Komponenten im HTML-Baum weit voneinander entfernt sind, stößt die Kommunikation über das HTML-Template schnell an seine Grenzen. Hierfür können geteilte Services verwendet werden, die per Dependency Injection in die beiden kommunizierenden Komponenten injiziert werden und die Kommunikation übernehmen. Wie dies funktioniert soll im nächsten Abschnitt kurz erläutert werden.

2.2.3 Dependency Injection und Services

Das Dependency Injection Entwurfsmuster dürfte den meisten Software-Entwicklern aus dem Backend-Bereich bekannt sein. Im Prinzip geht es darum die Abhängigkeiten einer Klasse nicht in der Klasse zu erzeugen sondern stattdessen von außen (idealerweise über den Konstruktor) in die Klasse zu injizieren (vgl. Starke, 2015, S. 77-78).

Dies sorgt dafür, dass die Abhängigkeiten explizit in der Konstruktorsignatur sichtbar sind, was einem Entwickler schneller den Überblick über den Kontext der

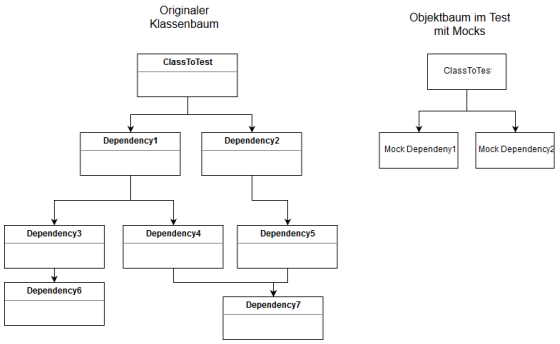

Abbildung 2.1: Klassenbaum und vereinfachter Objektbaum im Unit-Test mit Mocks

Klasse verschafft (vgl. Lilienthal, 2017, S. 72). Ein weiterer Vorteil ist die erhöhte Testbarkeit. Um einen Baustein zu testen, ist es möglich Mock-Objekte in den Konstruktor zu injizieren und das Verhalten des Bausteins isoliert zu testen und teure Operationen wie das Laden von Http-Resourcen im Test zu vermeiden. Dies vereinfacht auch das Herstellen eines Testszenarios, da nicht für den kompletten Baum Objekte instanziiert werden müssen, sondern lediglich für die Kinder der zu testenden Klasse (siehe Abbildung 2.1).

Dependency Injection wird in Angular konsequent umgesetzt. Hierfür wird im Framework der Begriff Service für alle Klassen verwendet, die per Dependency Injection injiziert werden können. Dieser Begriff stimmt nicht zwingend mit dem bekannten Service Begriff aus dem Domain Driven Design (DDD) (vgl. Starke, 2015, S. 84) überein, obwohl natürlich auch ein DDD-Service ein Angular Service sein kann. Technisch werden die Angular Services mit der @Injectable()-Annotation versehen und über ein Modul oder eine Komponente bereitgestellt (provided). Sie stehen dann anderen Komponenten und Services per Konstruktor-Injection zur Verfügung (vgl. Malcher u. a., 2017, S. 117-119). Es können nicht nur Klassen provided werden, sondern auch Factories oder konkrete Instanzen von Klassen.

Da die Services grundsätzlich als Singletons bereitgestellt werden (dies lässt sich bei Bedarf einstellen), ist es möglich mehreren Komponenten die selbe Service-Instanz zu injizieren. Über Service-Events und den geteilten Zustand des Services kann eine Kommunikation zwischen mehreren Komponenten aufgebaut werden, ohne dass diese sich explizit kennen müssen.

Bei Unit-Tests können über die TestBed-Klasse verschiedene Services und Service-Mocks provided werden, um Komponenten und Services isoliert zu testen. Technisch

wird dies zum Beispiel über das Providen einer speziellen Mock-Klasse oder Instanz realisiert.

```
TestBed.configureTestingModule({
  providers: [
    {
      provide: ExpensiveService,
      useClass: MockOfExpensiveService
    },
    {
      provide: AnotherExpensiveService,
      useValue: { loadEntities: () => mockedEntities }
    },
    ServiceToTest
  ]
});

let serviceInstance: ServiceToTest = TestBed.get(ServiceToTest);
// test the serviceInstance
```

Diese Flexibilität bei der Dependency Injection unterstützt eine saubere Strukturierung der Services und deren Abhängigkeit, sowie die Architektur von modularen und gut testbaren Komponenten und Services. Für eine Modularisierung auf höherer Ebene bietet Angular zusätzlich noch ein Modulkonzept, das im nächsten Abschnitt erläutert werden soll.

2.2.4 Module

Mit Modulen bietet Angular die Möglichkeit einer groben Strukturierung der Anwendung. Komponenten und Services können so (meist fachlich) zu größeren Bausteinen gruppiert werden. Technisch ist dies über die @NgModule Annotation gelöst, in der Imports, Exports, Services und Deklarierte Komponenten beschrieben werden. So kann ein Modul andere Module imporieren, Services providen und Komponenten entweder nur innerhalb des Moduls deklarieren, oder als Export auch anderen Komponenten zur Verfügung stellen (vgl. Malcher u. a., 2017, S. 301-308). Importiert ein Modul ein anderes Modul, stehen dem importierenden Modul automatisch alle exportierten Services und Komponenten des importierten Moduls zur Verfügung.

Ganz im Sinne des “Separation of Concerns”-Prinzips lassen sich so voneinander unabhängige Aufgabenbereiche in unterschiedliche Module verschieben. Durch die

Import- und Export-Beschreibungen am Modul wird die Schnittstelle der Module direkt auf oberster Ebene festgelegt.

Module können explizit bestimmte Komponenten nicht exportieren und sich so die Flexibilität bewahren, an diesen internen Komponenten beliebige Veränderungen vorzunehmen, solange die Komponenten an der Schnittstelle sich nach wie vor so verhalten wie zuvor. Dieses Prinzip der Minimalen Schnittstellen verringert die Kopplung und erleichtert es so, die Zusammenhänge in dem System zu überblicken (vgl. Lilienthal, 2017, S. 72-73).

Zyklische Abhängigkeiten zwischen den Modulen werden direkt vom Framework unterbunden und können nicht in die Anwendung eingeführt werden. Zudem sind die Beziehungen auf Modulebene direkt an den Modulen selbst sichtbar und somit explizit deklariert. Verletzungen werden hier zumindest auf Ebene der Komponenten und Services ebenfalls von Angular unterbunden.

Da ein Angular Modul lediglich aus einer Annotierten Typescript Klasse in einem Unterordner besteht, sind die so deklarierten Module sehr leichtgewichtig und ermöglichen eine viel feinere Modularisierung als zum Beispiel auf Backend Seite mit Maven, Gradle oder Visual Studio, wo jedes neue Modul bzw. Projekt einen deutlich höheren Overhead mit sich bringt, alleine schon durch die Ordnerstruktur. Während im Backend Bereich in der Regel einige wenige große Module erstellt werden, kann in Angular jedes dieser großen Module noch rekursiv in weitere kleinere Module unterteilt werden, ohne die Übersichtlichkeit und die Komplexität des Build-Prozesses negativ zu beeinflussen.

Im folgenden Beispiel einer solchen @NgModule()-Annotation wird das Modul ExampleModule beschrieben. Jedes Modul das das ExampleModule importiert erhält Zugriff auf die Services und auf die MainComponent, nicht aber auf die SubComponents oder auf die vom ExampleModule importierten Module, da diese nicht explizit exportiert werden. Die Komponentenstruktur kann also abgesehen von der MainComponent ohne Änderungen in importierenden Modulen angepasst werden.

```
1 @NgModule({
2   imports: [
3     CommonModule,
4     AnotherBaseModule
5   ],
6   providers: [
7     MainService,
8     AnotherService
```

```
  ],
  declarations: [
    MainComponent,
    SubComponent,
    AnotherSubComponent
  ],
  exports: [
    MainComponent
  ]
})
export class ExampleModule {
}
```

Durch dieses relativ mächtige und trotzdem leichtgewichtige Modulkonzept, wird das Entwickeln komplexer Anwendungen unterstützt, indem ein Top-Down Programmverstehen des Codes vereinfacht wird (.vgl Lilienthal, 2017, S. 81). Ein Entwickler muss nicht die Zusammensetzung und den Quellcode eines Moduls auf unterster Ebene verstehen, sondern kann sich auf das Verständnis der obersten Schnittstelle beschränken, solange diese konsequent umgesetzt ist. Hierbei spielt es keine Rolle ob das Modul aus vielen weiteren Modulen besteht, da die öffentliche Schnittstelle eines Moduls auch die benötigten Schnittstellen der Submodule mit einschließt.

2.2.5 RxJs Integration

Angular setzt für asynchrone und eventbasierte Kommunikation auf die Reactive Extensions für JavaScript (RxJs) 14. Diese ermöglichen das Schreiben von asynchronem Code und stellen hierfür eine Reihe von Funktionalität zur Verfügung, wie sie aus der funktionalen Programmierung bekannt ist. Kern bildet die Observable Klasse, die eine Quelle für ein Objekt oder eine Reihe von Objekten darstellt, die erst später verfügbar sein werden, (vergleichbar mit einem Promise oder Future). Hierbei kann es sich um das Ergebnis eines HTTP-Requests handeln oder um eine Quelle von Events. Da die APIs der Reactive Extensions in einigen Beispielen in dieser Arbeit verwendet werden, soll das Grundprinzip an dieser Stelle kurz erläutert werden.

Soll das Ergebnis eines solchen asynchronen Observables verwendet werden, wird mit der subscribe Methode eine Funktion (Callback) registriert, die ausgeführt werden soll, sobald (oder jedes Mal) wenn das Ergebnis eingetroffen ist. Zusätzlich

bietet die API die Möglichkeit, einen Error-Handler zu definieren und vielfältige Transformationen auf dem Observable auszuführen, bevor das Ergebnis vorliegt. Mit einer Vielzahl von funktionalen Operationen kann das Observable in ein Observable eines anderen Typs umgewandelt werden (z.B. der JSON-Body einer HTTP-Response in eine Instanz der entsprechenden fachlichen Klasse). Es gibt auch die Möglichkeit, Observables zu kombinieren, zu filtern, Ergebnisse zu puffern usw.

```
observable.map(response => this.mapToEntity(response))
          .filter(entity => entity.isActive)
          .subscribe(entity => this.onEntityLoaded(entity),
                     error => this.onErrorOccurred(error));
```

Hierbei wird technisch an dem Observable Stück für Stück mehr Code hinterlegt der beim Eintreffen des asynchronen Ergebnisses ausgeführt wird, das Ergebnis dann entsprechend transformiert und an die Subscriber verteilt. Dies ermöglicht das Programmieren mit asynchronem Code als wäre das Ergebnis schon vorhanden. Technische Services können die technischen Transformationen auf den Observables vornehmen und den fachlichen Services dann Observables mit fachlichen Enitäten als Rückgabewert bereitstellen. Melden sich die fachlichen Services dann per subscribe an dem Observable an, werden erst die technischen Schritte aus dem technischen Service ausgeführt, ohne dass der fachliche Service davon etwas merkt. Sobald diese Transformationen abgeschlossen sind, wird der Callback im fachlichen Service ausgeführt.

In Angular liefert zum Beispiel der HttpClient Observables zurück. Ein weiterer Anwendungsfall sind die EventEmitter (die von der Observable Klasse erben) und eingesetzt werden um Events aus den Komponenten zu verteilen und das Eventbinding von Angular zu ermöglichen.

Dieses Prinzip ermöglicht ein sehr komfortables Arbeiten mit asynchronem Code, da die Rückgaben der Observables an der Stelle im Code bearbeitet werden können, an der sie auch angefordert wurden, obwohl das Ergebnis zum Zeitpunkt der Bearbeitung noch gar nicht vorliegt.

2.2.6 Weitere Konzepte

Der Vollständigkeit halber seien hier noch die Konzepte von Pipes und Direktiven erwähnt. Da diese jedoch im weiteren Verlauf der Arbeit nicht aufgegriffen werden, wird an dieser Stelle darauf verzichtet genauer darauf einzugehen.

3 Anwendung der Konzepte

In diesem Abschnitt werden verschiedene Möglichkeiten und Vorgehensweisen beschrieben, die für das Entwickeln einer geteilten Basis und erweiternder Anwendungen hilfreich sind. Zunächst geht es darum, wie mehrere Anwendungen parallel entwickelt werden können und welche Möglichkeiten das Angular CLI hierfür bietet. Dann sollen mehrere hilfreiche Konzepte anhand von kompakten Beispielen erläutert werden. Hierbei geht es um die Verwendung von Basiskomponenten durch Vererbung und Templatebasierter Konfiguration, das Providen von speziellen Serviceausprägungen, das Transclusion Prinzip von Angular sowie einige Möglichkeiten zum Bereitstellen von Extension-Points.

3.1 Multi-App Setup

Um zwei Apps zu entwickeln die auf einer gemeinsamen Komponente aufsetzen, muss zunächst das Angular CLI so konfiguriert werden, dass mit mehreren Apps gleichzeitig entwickelt werden kann.

Ab der CLI Version 6.X.X, die mit dem Angular 6 Release die 1.7.X Versionen ablöst, gibt es einige Grundlegende Änderungen. So ändert sich die Syntax der .angular-cli.json, welche ab Version 6 angular.json heißt und es gibt einen verbesserten Support für das Erstellen mehrerer Apps und für das Erstellen von Libraries.

Da nicht in jedem Projekt die Möglichkeit eines sofortigen Updates auf Angular 6 besteht, soll im nächsten Abschnitt zunächst das Vorgehen mit der alten Version des Angular CLI beschrieben werden (< 6.0.0). Ein späteres Update dieser Konfiguration auf Angular 6 ist mit dem ng update Befehl des neuen Angular CLI problemlos möglich. Lediglich der Library-Support muss dann noch von Hand nachkonfiguriert werden.

Im darauf folgenden Abschnitt 3.1.2 wird das vereinfachte Vorgehen mit den neuen Features des Angular CLI beschrieben. Da diese Features eine deutlich bessere

Unterstützung des Multi-App Setups bieten, ist die Konfiguration mit dem Angular CLI 6 vorzuziehen.

3.1.1 Multi-App Setup bis Angular Version 5

Grundsätzlich wird eine Strukturierung in zwei Apps (app1 und app2) sowie eine Basiskomponente (core) angestrebt. Ziel ist es, dass nach dieser initialen Konfiguration das Entwickeln an dem Multi-App-Projekt genauso funktioniert, wie mit einer einzigen App.

Die Angestrebte Ordnerstruktur ist in Abbildung 3.1 zu sehen. Die root-Ordner der beiden Apps enthalten jeweils eine AppComponent sowie ein AppModule als Basis wie aus Single-App-Projekten bekannt. Hinzu kommt ein eigener Asset-Ordner sowie die main.ts und test.ts Dateien die für das Bootstrapping und die Testausführung nötig sind. Außerdem sollten die Apps jeweils eine eigene environments-Konfiguration besitzen.

Um das `ng generate`-Feature des CLIs weiter problemlos nutzen zu können, hat es sich als sinnvoll erwiesen, den src/app Ordner als root-Ordner zu benutzen und erst in diesem die Aufteilung der Apps vorzunehmen.

Ausgehend vom Ordner app1 oder app2 soll das Entwickeln so funktionieren wie in einem Single-App-Projekt ausgehend vom Ordner app.

Module und Ordnerstruktur

Der erste Schritt ist, wie bei Angular Projekten üblich, das Erzeugen eines Boilerplate-Projektes mit dem Angular CLI per `ng new <projektname>`. Dies erzeugt im Ordner src/app eine einziges Modul, was für die weiteren Anforderungen nicht ausreichend ist.

Es wird also die in Abbildung 3.1 beschriebene Ordnerstruktur aufgesetzt und das Basismodul und die Basiskomponente in beide Unterordner verschoben bzw. kopiert. Gleiches gilt für die main.ts. Hier muss beim Korrigieren der Imports darauf geachtet werden, dass das korrekte AppModule importiert wird und nicht das der anderen App. Auch der environments Ordner wird jeweils einmal pro App in die entsprechenden Ordner verschoben, um eine separate Konfiguration der Apps zu ermöglichen.

Zuletzt kann mit dem Befehl `ng g module core` das Core-Modul generiert werden welches dann von den beiden App-Modulen importiert werden muss.

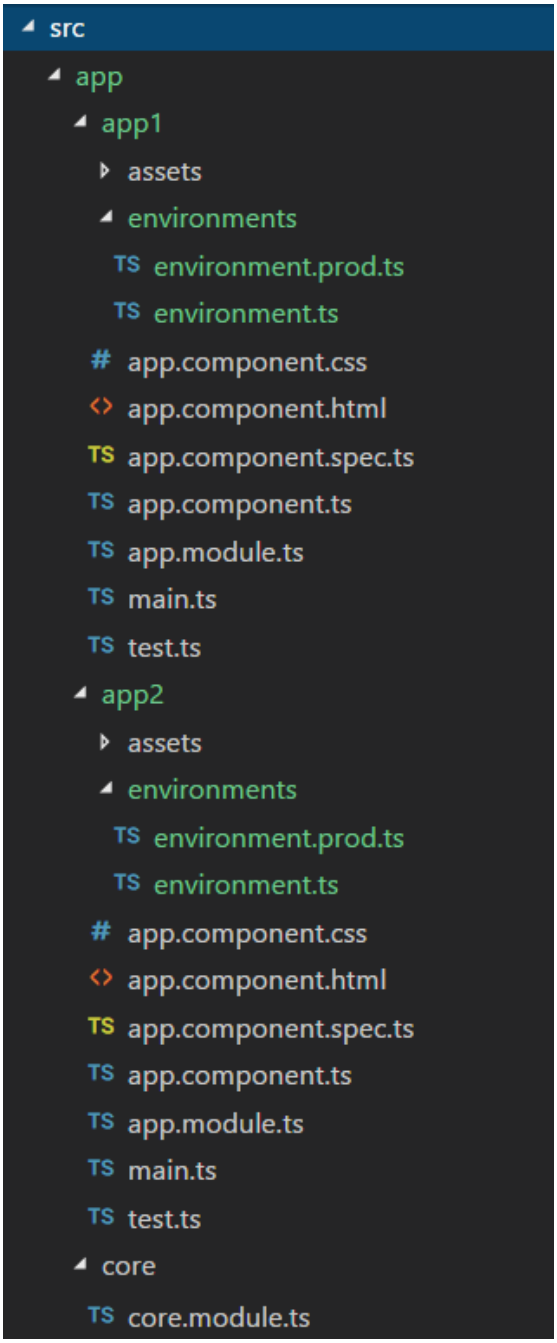

Abbildung 3.1: Ordnerstruktur mit root-Modulen und Components

Hiermit ist die Ordnerstruktur abgeschlossen, damit das Projekt benutzbar wird sind jedoch noch einige weitere Schritte nötig.

Zweite App einrichten

Das Angular CLI bietet einige Möglichkeiten um die Entwicklung mehrerer Apps im selben Projekt zu unterstützen. Dies ermöglich das Entwickeln der Basiskomponente in einer IDE parallel zu den beiden darauf aufsetzenden Anwendungen. Diese Funktionalität ist in dem Angular CLI Wiki dokumentiert 3 und ermöglicht das abwechselnde oder sogar parallele Starten, Bauen und Testen mehrerer Angular Apps im selben Projekt.

Dies geschieht über das Anpassen der Konfigurationsdatei .angular-cli.json.

Hierzu wird zunächst der vom Angular CLI bereits vorgenerierte Eintrag unter apps kopiert, so dass das apps-Array nun zwei identische Einträge enthält, einen pro App. Um sinnvoll mit diesen beiden Apps arbeiten zu können sind noch einige weitere Konfigurationen erforderlich die im Folgenden erläutert werden sollen.

So ist es wünschenswert, dass die verschiedenen Apps per Name angesprochen werden können, für das Bootstrapping unterschiedliche Angular Components verwenden, separate Asset-Ordner nutzen und in verschiedene Output-Ordner gebaut werden, um einen parallelen Betrieb sinnvoll zu ermöglichen.

Apps benennen

Den Apps kann durch das Hinzufügen eines name-Attributes ein Name zugewiesen werden. Dies sorgt dafür, dass die Befehle des Angular CLIs an eine spezielle App Adressiert werden können. Per Default werden die Befehle (wie `ng serve` und `ng build`) immer an die erste App in dem apps-Array gerichtet.

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "project": {
    "name": "client"
  },
  "apps": [
    {
      "name": "app1",
      ...
    },
```

```
    {
      "name": "app2",
      ...
    }
  ],
  ...
}
```

Listing 3.1: .angular-cli.json

Durch das Hinzufügen des Parameters `--app=<app-name>` zu einem der `ng` Befehle, richtet sich der Befehl an die App des übergebenen Namens. So kann mit `ng serve --app=app1` die eine und mit `ng serve --app=app2` die andere App gestartet werden.

Um dieses Verhalten noch etwas komfortabler in den Entwicklungs- bzw. Buildprozess zu integrieren, kann in der package.json-Datei noch der **scripts** Eintrag angepasst werden um die Befehle direkt per npm npm (2018) auszuführen. Zum Beispiel so wie in dem Folgenden Auszug:

```
{
  "name": "multi-app-client",
  "version": "1.0.0-SNAPSHOT",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start1": "ng serve --app=app1",
    "start2": "ng serve --app=app2 --port=4201",
    "build1": "ng build --prod --app=app1",
    "build2": "ng build --prod --app=app2",
    "test1": "ng test --app=app1",
    "test2": "ng test --app=app2",
  },
  ...
}
```

Listing 3.2: package.json

Dies ermöglicht durch die Konfiguration des Ports bei dem `start2` Befehl die parallele Ausführung beider Anwendungen per `npm run start1` und `npm run start2`. Die erste App läuft dann auf dem Default-Port 4200, die zweite auf dem Port 4201.

Anpassen der Pfade

Als nächster Schritt werden die Pfade zu main.ts, test.ts, outDir sowie zu den environments wie in im Folgenden angepasst. Um den `ng g` Befehl weiter nutzen zu können, wird der root-Eintrag nicht verändert.

```
{
  ...
  "apps": [
    {
      "name": "app1",
      "root": "src",
      "outDir": "dist/app1",
      "main": "app/app1/main.ts",
      "test": "app/app1/test.ts",
      "environmentSource": "app/app1/environments/environment.ts",
      "environments": {
        "dev": "app/app1/environments/environment.ts",
        "prod": "app/app1/environments/environment.prod.ts"
      },
      ...
    },
    {
      "name": "app2",
      "root": "src",
      "outDir": "dist/app2",
      "main": "app/app2/main.ts",
      "test": "app/app2/test.ts",
      "environmentSource": "app/app2/environments/environment.ts",
      "environments": {
        "dev": "app/app2/environments/environment.ts",
        "prod": "app/app2/environments/environment.prod.ts"
      },
      ...
  ],
  ...
}
```

Listing 3.3: .angular-cli.json

Mit dieser Konfiguration werden die beiden Apps in separate Ordner gebaut und beim Bootstrapping wird durch die passende main.ts-Datei das zugehörige

AppModule geladen. Da die Template Direktiven der Root-Module identisch sind, kann die gleiche index.html Datei verwendet werden.

Separate Asset Ordner

Ziel dieser Konfiguration ist es, dass die beiden Apps jeweils einen eigenen unabhängigen Asset-Ordner besitzen, die Assets im Code aber trotzdem per `assets/<dateiname>` erreichbar sind. Hierfür ist eine erweiterte Asset Konfiguration nötig, wie im Angular CLI Wiki dokumentiert 2.

```
{
  ...
  "apps": [
    {
      "name": "app1",
      ...
      "assets": [
        { "glob": "**/*",
          "input": "./app/app1/assets/",
          "output": "./assets/"
        }
      ],
      ...
    },
    {
      "name": "app2",
      ...
      "assets": [
        {
          "glob": "**/*",
          "input": "./app/app2/assets/",
          "output": "./assets/"
        }
      ],
      ...
    }
  ],
 ...
```

```
29 }
```

Listing 3.4: .angular-cli.json

Diese Konfiguration sorgt dafür, dass die Assets aus dem jeweiligen Assert Ordner beim Build in den Ordner assets kopiert werden, wo sie so wie in Single-App-Projekten referenziert werden können.

Fertig konfiguriertes Basisprojekt

Diese Basiskonfiguration erlaubt nun das parallele Entwickeln mit zwei Apps ohne das eine Umgewöhnung vom bekannten Vorgehen nötig ist.

So kann ein neues Modul für die erste App per `ng g module app1/<modul-name>` generiert werden, Komponenten darin per
`ng g module app1/<modul-name>/<componenten-name>` usw.

Da diese Vorgehensweise recht aufwändig ist und viele manuelle Schritte erfordert, wird im nächsten Abschnitt das deutlich einfachere Vorgehen mit Angular 6 beschrieben.

3.1.2 Multi-App Setup ab Angular Version 6

Mit der Version 6 des Angular CLI gibt es deutlich verbesserten Support für das Arbeiten mit mehreren Apps und das Erstellen von Libraries. So ist es jetzt mit ng generate application <app-name> und ng generate library <library-name> möglich, Anwendungen und Libraries direkt zu generieren. Diese landen per Default im Ordner projects und werden direkt in der angular.json Datei (vorher .angular-cli.json) eingetragen. Diese Ordnerstruktur unterscheidet sich von der Ordnerstruktur des CLI mit Version 1.7.X und ist noch besser auf das Arbeiten mit mehreren Anwendungen und Libraries zugeschnitten (siehe Abbildung 3.2). Ein ng-Befehl kann in der neuen CLI-Version durch das Anhängen des Namens der Anwendung oder Library an die entsprechende App/Library gerichtet werden (`ng serve <app-name>`,`ng build <library-name>`, ...). Beim generieren von Modulen, Komponenten und Service muss das Projekt mit einem extra Parameter angegeben werden (`ng g component <component-name> --project <library-name>`,
`ng g module <module-name> --project <app-name>`).

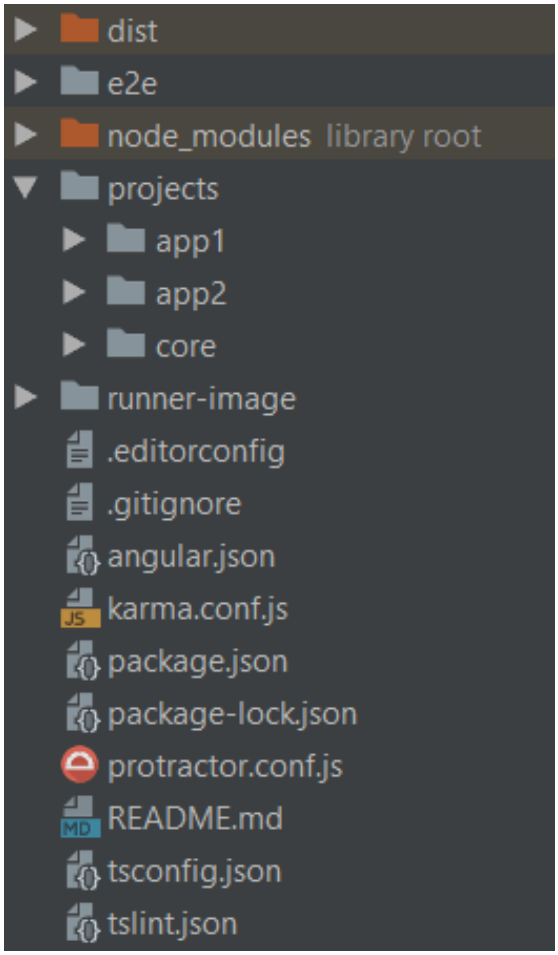

Abbildung 3.2: Neue Ordnerstruktur im CLI 6

Grundlagen Library Support

Da weitere Anwendungen wie im vorigen Kapitel (3.1.1) beschrieben ohne großen Aufwand manuell hinzugefügt werden konnten, stellt der Library Support die größere Neuerung des CLI dar.

Der Befehl ng generate library <library-name> ermöglicht das Erstellen einer Library die per ng build <library-name> gebaut werden kann. So kann das core-Modul in eine eigenständige Library ausgegliedert werden, mit einer explizit definierten öffentlichen API, Imports ohne direkte Dateipfade und der Möglichkeit die Library auf npm npm (2018) zu veröffentlichen.

Die interne Ordnerstruktur einer solchen Library (Abbildung 3.3) ähnelt der einer Angular-App. Neben den Konfigurationsdateien für Typescript, Karma und Angular selbst (ng-package.json) gibt es einen src-Ordner. Styles, Polyfills, index.html und Ähnliches gibt es hier nicht, da die Library nicht direkt als html-Seite gebaut wird. Stattdessen gibt es eine public_api.ts-Datei, die die öffentliche API der Anwendung spezifiziert. Mehr dazu im nächsten Abschnitt (3.1.2).

Der folgende Abschnitt kann im Detail auf der entsprechenden Seite des Angular-CLI Wikis nachgelesen werden (vgl. Larsen, 2018).

Die erste wichtige Limitierung einer Angular Library ist die Notwendigkeit, die Library nach jeder Änderung neu zu bauen damit die benutzenden Apps diese Änderungen bemerken. In zukünftigen CLI Versionen ist ein watch-Support für Libraries geplant um diesen Nachteil etwas auszugleichen, in der Version 6.0.3 der CLI ist dies aber noch nicht der Fall.

Bei der Entwicklung von Anwendungen mit einer geteilten Basiskomponente, stellt sich also die Frage, ob das Basismodul von Anfang an in eine Library auslagert und der Produktivitätsverlust durch die regelmäßigen Builds der Library in kauf genommen werden, oder ob die Basiskomponente zunächst in einer der Anwendungen zu verort und später in eine Library extrahiert wird.

Gerade beim Entwickeln der ersten Applikation wird die Basis sehr häufigen und auch grundlegenden Änderungen unterworfen sein, da sie parallel mit der Anwendung entsteht. In aller Regel gibt es zu diesem Zeitpunkt auch noch keine zweite Anwendung, das Auslagern der Basis geschieht auf Vorrat um beim Entwickeln der zweiten Anwendung schneller zu sein und eine besser Strukturierte Anwendung zu entwickeln.

In diesem Fall bietet es sich an, während der ersten Entwicklungsphase nur eine einzige Anwendung zu generieren (diese landet im Ordner projects/<app-name>),

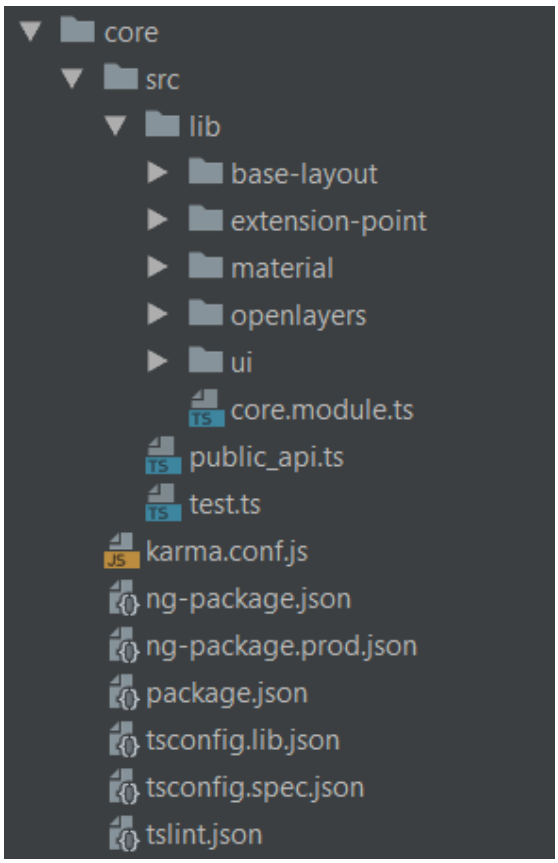

Abbildung 3.3: Ordnerstruktur einer Angular Library (Screenshot aus der Beispielanwendung)

das Erstellen der Library auf später zu verschieben und zunächst in einem Modul in der Anwendung zu verorten. Hierbei ist es natürlich wichtig die beiden Module gut voneinander abzugrenzen, damit die Migration des Basismoduls in die Library später problemlos funktioniert.

Explizite Definition der öffentlichen Library-API

Einer der wesentlichen Unterschiede die das Arbeiten mit einer Library mit sich bringt, ist die Definition einer öffentlichen API und dadurch die Struktur der Imports.

Wird eine Klasse aus einem anderen Modul in der selben Applikation importiert, handelt es sich um einen relativen Import der exakten Datei, in der die Klasse deklariert wird.

Ein solcher Import kann beispielsweise folgendermaßen aussehen:

```
1 import {ClassToImport} from '../../core/module1/component1/class-to-import';
```

Mit einer Library (im folgenden Beispiel core genannt) ändert sich dieser Import auf die folgende deutlich kompaktere Zeile:

```
1 import {ClassToImport} from 'core';
```

Um dies zu ermöglichen, muss die öffentliche API der core-Library klar definiert sein. Die CLI generiert hierfür eine Datei public_api.ts in der die Dateien explizit exportiert werden müssen, deren Klassen Teil der API der Library sein sollen. In der Praxis bietet es sich an, eine solche Datei für jedes Modul zu erstellen, aus dem Klassen öffentlich verfügbar sein sollen, und diese in der nächsthöheren public_api.ts Datei zu referenzieren.

```
1 export * from './lib/core.service';
2 export * from './lib/core.module';
3 export * from './lib/extension-point/public_api'
4 export * from './lib/openlayers/public_api'
```

Listing 3.5: public_api.ts

Ein solcher Eintrag in einer public_api-Datei ist also für jeder Klasse nötig, die in einer benutzenden Anwendung importiert werden sollen.

Bei einer Migration zu einer Angular Library müssen also zunächst alle öffentlichen Klassen explizit exportiert werden und dann alle Imports in der benutzenden Anwendung auf das einfachere Import-Schema (ohne relative Dateipfade) umgestellt werden. Damit die Library nach dem Bauen von den Apps erkannt wird, muss in

der tsconfig.json im Root des CLI-Workspaces (also im selben Verzeichnis wie der projects-Ordner) der Pfad zum dist-Verzeichnis der Library eingetragen werden. Diese Änderungen wird auch in der Applikation wirksam, da die tsconfig.json in der Applikation von der tsconfig-Datei im Root erbt.

```
{
  ...
  "compilerOptions": {
    ...
    "baseUrl": "./",
    "paths":  {
      "<library-name>": ["dist/<library-name>"]
    }
  }
}
```

Listing 3.6: tsconfig.json

Nun kann die Library referenziert werden, ohne auf npm veröffentlicht und installiert worden zu sein. Sie muss jedoch bei jeder Änderung neu gebaut werden, damit die Änderung in den benutzenden Anwendungen reflektiert wird.

Diese Einschränkung der öffentliche API bringt neben der deutlich kompakteren Imports vor allem den Vorteil, dass nicht nur auf Komponenten- sondern auch auf Klassenebene die Schnittstelle der Basis eingeschränkt werden kann. Services die intern für die Module der Basis-Library provided wurden, können durch das Weglassen der Export-Deklaration zu internen Services der Basis-Library gemacht werden. In der Library sind sie so für jeden Verfügbar, der das entsprechende Modul importiert, die benutzenden Anwendungen wissen jedoch nicht von der Existenz dieses Services. So kann der Service innerhalb der Library verändert oder ersetzt werden ohne dass hierfür Änderungen an den benutzenden Anwendungen nötig wären. Es empfiehlt sich daher, die öffentliche Schnittstelle der Library möglichst schlank zu halten. Auf diese Weise bleibt die Library flexibel und lässt intern größere Umstrukturierungen zu, ohne Breaking-Changes herbeizuführen.

Ausblick

In Zukunft ist davon auszugehen, dass der Library Support in Angular noch weiter verbessert wird. So ist unter anderem ein watch-Modus geplant, der den manuellen Build der Library ersetzt. So sollen in einem CLI-Workspace beim build (oder serve)

einer App/Library automatisch auch alle Dependencies dieser App/Library gebaut werden. Dies wird perspektivisch dazu führen, dass das Arbeiten mit Libraries in Angular noch komfortabler möglich sein wird.

Nach diesem technischen Exkurs in die Interna des Angular CLI sollen in den folgenden Abschnitten auf Architekturmuster und Prinzipien eingegangen werden, die hilfreich sind um Teile der Anwendungen in die gemeinsame Library auszulagern.

3.2 Basisservices

Die einfachste Möglichkeit Funktionalität in eine gemeinsame Basis auszulagern, ist das Schreiben von einzelnen Bausteinen, die durch Konfiguration, Komposition oder Vererbung auf die jeweilige spezielle Anwendung zugeschnitten werden, oder einfach direkt benutzt werden können. Dies geschieht bei einer guten Architektur automatisch und sollte stets das Ziel sein, um die Wiederverwendbarkeit des Codes zu erhöhen. Werden kombinierbare Bausteine auf verschiedenen Ebenen bereitgestellt, kann eine verwendende Klasse oder Anwendung auf verschiedenen Ebenen auf die Bausteine zugreifen und sie so anordnen, konfigurieren und erweitern, dass sich optimalerweise viele der Probleme der verwendenden Klasse oder Anwendung lösen lassen, ohne das Rad neu erfinden zu müssen.

3.2.1 Grundlagen Basisservices

Eine Art solcher Bausteine sind abstrakte Basisservices, die technische Details von der Fachlichkeit unabhängig behandeln. Hierbei können das generische Service Interface und eine generische Basisimplementierung in der Basis liegen. Die überschreibenden Klassen können dann z.B. per Hook-Methoden (auch Template-Methoden genannt) Gamma u. a. (1995)[vgl.][S. 360-365] die Funktionalität hinzufügen, die für die speziellen Entitäten und die jeweilige Fachlichkeit erforderlich ist. Da das generische Interface in der core Komponente verortet ist, können weitere speziellere Basisservices darauf Bezug nehmen. So können viele technische Details in der Basiskomponente implementiert werden, um technische Fehlerquellen und Overhead in den erweiternden Anwendungen zu reduzieren. Dieses Prinzip ist unter Anderem als “Single Point of Truth” bzw. “Don’t Repeat Yourself” bekannt (vgl. Starke, 2015, S. 69-70).

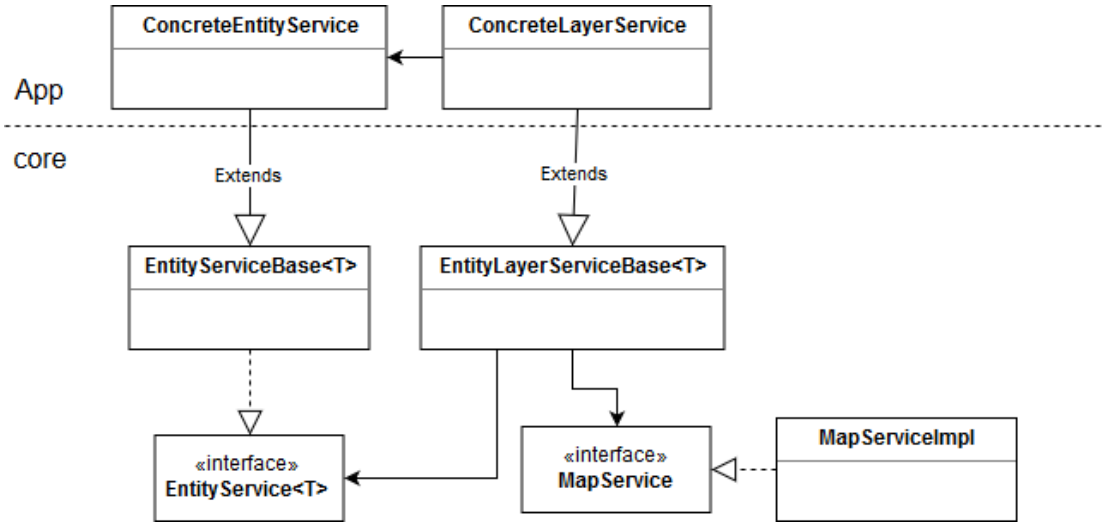

Abbildung 3.4: Beispiel einer Basisservice Struktur

3.2.2 Beispiel Basisservice Struktur

Abbildung 3.1 zeigt eine solche Struktur aus einer Kartenanwendung, die georeferenzierte Daten visualisiert (die Anwendung an sich ist im Abschnitt 3.6 noch einmal detaillierter beschrieben). Im konkreten Beispiel gibt es einen Server, der per ReST und Stomp via Websocket die Daten der Entitäten und Update-Events zur Verfügung stellt. Aufgabe des EntityServices ist es, hiervon zu abstrahieren und diese Dienste anderen Services und Komponenten zur Verfügung zu stellen. Die Entitäten werden hierfür vom Server per JSON serialisiert und so an den Client geschickt. Für die asynchronen Events wird clientseitig die Observable Klasse der Reactive Extensions für JavaScript (RxJs) verwendet.

Es ergibt sich folgende einfache Schnittstelle:

```
1 getAll(): T[];
2 changes(): Observable<ChangeEvent<T>>;
```

Da die Basisklasse keine Kenntnis der konkreten Entitäten hat, muss das Mapping der JSON-Objekte auf konkrete Typescript Klassen in einer abstrakten Hook-Methode geschehen, die von der erweiternden Klasse überschrieben werden muss. Urls und die benötigten Http- und Stomp-Services werden über den Konstruktor übergeben. Eine erweiternden Klasse muss also nur die Mapping-Methode überschreiben und die Urls und Services übergeben.

Der EntityLayerServiceBase sorgt auf Basis eines solchen EntityServices (und eines MapServices der die Kartenfunktionalität abstrahiert) dafür, dass die Entitäten auf der Karte dargestellt werden und jedes Update sofort sichtbar ist. Zudem sollen Filter gesetzt werden können, die Teile der Entitäten von der Karte entfernen.

Die Basisimplementierung übernimmt hierbei das Reagieren auf Change-Events, das Setzen der Filter, sowie eine Clustering Funktion der Features. Mit Features ist in diesem Kontext die in Geo-Anwendungen übliche Bezeichnung für Geometrie auf der Karte gemeint. Zudem gibt sie der erweiternden Klasse Zugriff auf die Selektionsevents des Kartenframeworks. Durch Hook-Methoden delegiert sie nur das Erzeugen der Features anhand einer Entität (und somit die Darstellung auf der Karte) an die erweiternde Klasse.

Eine möglicher Basisservice ist im folgenden Listing (deutlich vereinfacht) skizziert.

```
1
2  export abstract class EntityLayerServiceBase<TEntity, TFeature> {
3          private layer: FeatureLayer;
4          ...
5
6          constructor(
7                  protected entityService: EntityService<TEntity>,
8                  protected mapService: MapService) {
9
10                 this.layer = this.mapService.createFeatureLayer();
11                 ... // layer configuration
12                 this.mapService.addLayer(this.layer);
13
14                 this.layer.featureSelected
15                         .subscribe(f => this.onFeatureSelected(f as any));
16                 this.entityService.changes()
17                         .subscribe(change => this.refreshMap());
18         }
19
20         private refreshMap() {
21                 let features: EntityFeature[] = this.entityService.getAll()
22                         .map(e => this.mapToFeature(e));
23                 this.layer.clear();
24                 this.layer.addFeatures(features);
25         }
```

```
26
27          protected abstract mapToFeature(entity: Entity): EntityFeature;
28
29          protected abstract onFeatureSelected(feature: EntityFeature);
30          ...
31  }
```

Listing 3.7: entity-layer-service-base.ts

Eine Erweiterung kann dann folgendermaßen aussehen:

```
1  @Injectable()
2  export class ConcreteEntityLayerService
3          extends EntityLayerServiceBase<Entity, EntityFeature> {
4
5          constructor(
6                  entityService: EntityService<Entity>,
7                  mapService: MapService) {
8
9                  super(entityService, mapService);
10         }
11
12         protected mapToFeature(entity: Entity): EntityFeature {
13                 return new EntityFeature(entity);
14         }
15
16         protected onFeatureSelected(feature: EntityFeature) {
17                 // selection handling
18         }
19 }
```

Listing 3.8: concrete-entity-layer-service.ts

Obwohl dieses Beispiel nicht Angular-spezifisch ist, demonstriert es doch, wie durch Typescripts Unterstützung von Vererbung bekannte Muster aus dem Backend-Bereich auch in einem Angular-Frontend umgesetzt werden können.

3.3 Providen von speziellen Serviceausprägungen

Das Providen von speziellen Serviceausprägungen stellt eine weitere Möglichkeit dar, wie das Verhalten von einem Basismodul durch erweiternde Module angepasst

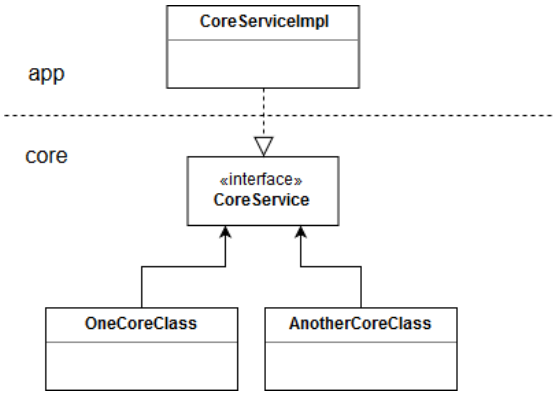

Abbildung 3.5: Service Implementierung in erweiternder Anwendung

werden kann. Zudem wird durch die Kontrolle, die eine erweiternde Anwendung über den neuen Service hat, eine bessere Integration mit dem Basismodul möglich.

3.3.1 Grundprinzip Serviceausprägungen

In der Basiskomponente wird zunächst ein Service Interface deklariert. Dieses Interface kann von beliebigen Services und Komponenten der Basiskomponente per Dependency Injection injiziert und benutzt werden, ohne dass eine Implementierung in der Basiskomponente existiert. Um die Anwendung dann benutzbar zu machen, ist es Aufgabe der erweiternden Anwendung, das Service Interface zu implementieren und zu providen, so dass alle Basisklassen die eine Abhängigkeit von dem Service-Interface besitzen, die spezielle Serviceausprägung der erweiternden Klasse injiziert bekommen 3.5.

3.3.2 Beispiel Serviceausprägungen

Ein Beispiel für eine solche Mechanik ist der Clientseitige Service für Serverevents, im Falle der Beispielanwendung per Stomp. Das ServerEventService-Interface besteht lediglich aus einer Methode, die zu einem Topicnamen einen Stream von Textnachrichten als String liefert.

```
1  getTopic(topicName: string): Observable<string>;
```

Services die den ServerEventService benutzen wollen, müssen keine Kenntnisse über den Server, URLs oder die Implementierung des Messaging haben, sie müssen nur den Namen des Topics kennen für das sie sich registrieren wollen, und die

Struktur der Nachricht vom Server um diese zu parsen und auf Typescript Klassen zu mappen.

```
export class EntityService {
  constructor(private serverEventService: ServerEventService,
              ...) {
  }

  public changes(): Observable<Entity> {
    return this.serverEventService.getTopic('entity-topic-name')
              .map(message => JSON.parse(message))
              .map(this.mapToEntity);
  }

  private mapToEntity = (dto: any): Entity => {
    ...
  }
  ...
}
```

Die erweiternde Anwendung muss nur das ServerEventService Interface implementieren und providen.

```
@NgModule({
  providers: [
    {
      provide: ServerEventService,
      useClass: ServerEventServiceImpl
    },
    ...
  ],
  ...
})
```

Alle Details von der verwendeten Library (stompjs, ActiveMQ) über die URLs der Websocket Endpunkte, bis hin zu Security Lösungen z.B. in Stomp-Headern werden in der erweiternden Anwendung umgesetzt, ohne dass die Basis hiervon etwas wissen muss. So kann jede Anwendung das Messaging eigenständig umsetzen und trotzdem auf die Services der Basis zurückgreifen.

Nach diesem Prinzip können Entscheidungen über die spezifische Implementierung eines Services außerhalb der Basis getroffen werden und machen diese so

flexibler. Solange die Verträge der Service-Interfaces eingehalten werden, kann jede Anwendung auf die eigenen speziellen Bedürfnisse zugeschnittene Implementierungen providen, die von den Basisservices problemlos genutzt werden können.

3.4 Transclusion

In den vorigen Abschnitten wurde deutlich, dass das Bereitstellen von konfigurierbaren und erweiterbaren Bausteinen eine sehr flexible Gestaltung der erweiternden Anwendungen möglich macht. Dies kann je nach Anwendungsfall sehr positiv sein, birgt aber auch Nachteile.

Zum einen wird durch die vielen Möglichkeiten die Komplexität in den erweiternden Anwendungen erhöht, zum anderen haben die Anwendungen (speziell im Bezug auf die Oberfläche) nicht zwingend den selben Aufbau. Dies kann für sehr individuelle Erweiterungen der Basis von Vorteil sein, üblicherweise ist jedoch in gewissem Maße gewünscht dass die Anwendung einen Wiedererkennungswert hat. Hier kann es helfen, in der Basisanwendung bereits eine Grundstruktur der Oberfläche vorzugeben, in die dann die Komponenten der erweiternden Anwendungen eingefügt werden können. Eine Lösung die Angular für dieses Problem bietet heißt Transclusion und soll in diesem Abschnitt erläutert werden.

3.4.1 Grundprinzip Transclustion

Das Grundprinzip von Transclusion ist denkbar einfach: Über die Direktive `<ng-content></ng-content>` wird eine Stelle im Template einer Komponente markiert, an die eine andere Komponente gerendert werden soll. In der benutzenden Komponente wird die Direktive der einzufügenden Komponente einfach zwischen die Tags der Komponente geschrieben, die das `<ng-content></ng-content>` im Template enthält (siehe 3.4.2).

Im Folgenden soll an einigen Beispielen gezeigt werden, wie Transclusion in einer Basiskomponente genutzt werden kann.

3.4.2 Einfaches Transclusion Beispiel

So ist es zum Beispiel möglich eine Komponente mit Header und Footer zu erzeugen und den Content über Transclusion einzufügen.

```
1 <div>
```

```
  ... // header code
  <ng-content></ng-content>
  ... // footer-code
</div>
```

Im benutzenden Code kann dann die Komponente wie folgt benutzt werden:

```
<transclusion-component>
  ... // page content
</transclusion-component>
```

So kann die selbe Struktur mit Header und Footer mehrfach in der Anwendung benutzt werden, als Single Point of Truth steht jedoch nach wie vor das erste Template zur Verfügung. Jede Änderung am Template dieser Komponente wird direkt in allen benutzenden Komponenten reflektiert.

Das nächste Beispiel soll eine Anwendungsmöglichkeit für eine Basiskomponente demonstrieren.

3.4.3 Beispiel Grundlayout

Eine Möglichkeit ist die Definition eines Grundlayouts. Hierdurch können die Freiheiten der erweiternden Apps eingeschränkt und zugleich die Sichtbarkeit der in der Basis verwendeten Komponenten reduziert werden. Wenn diese nicht im Basismodul exportiert werden müssen, führt das zu einer weiteren Entkopplung die dafür sorgt, dass die Grundstruktur der Basis verändert werden kann, ohne die erweiternden Apps anzupassen (vgl. Starke, 2015, S. 68). Insgesamt bietet die Basiskomponente so eine schlankere und besser definierte Schnittstelle.

Als Beispiel soll das in Abbildung 3.6 zu erkennende Layout dienen. Hier gibt es eine Hintergrundkarte, über der oben links in der Ecke ein Logo zu sehen ist, rechts und links am Rand ist jeweils ein Container der eine Komponente beinhaltet, die aus der erweiternden Anwendung stammt. Die URL der Hintergrundkarte und das Logo sollen konfigurierbar sein.

Zunächst ist das Beispiel ohne Transclusion umgesetzt:

```
<app-logo [src]="'assets/ecorp.jpg'"></app-logo>
<app-ol-map [mapUrl]="mapUrl"></app-ol-map>

<app-right class="right-container"></app-right>

```

Abbildung 3.6: Basislayout für das Transclusion Beispiel

```
6 <app-left class="left-container"></app-left>
```

Listing 3.9: layout-no-transclusion.html

Die erweiternde Komponente kennt in diesem Beispiel sowohl die Logo- als auch die Map-Component. Die `<app-left>` und `<app-right>` Direktiven stammen hier von Komponenten aus der erweiternden App.

Die Container app-left und app-right müssen mit CSS-Klassen in der erweiternden Komponente positioniert werden, während das Logo vom Basis-Modul positioniert wird. Dies führt dazu dass das Basis-Modul die Position und Größe des Logos nicht mehr problemlos ändern kann, ohne dass Änderungen in den erweiternden Apps nötig sind. Kommt Funktionalität zum Schließen und erneutem Öffnen der beiden Seitentabs hinzu, so muss diese redundant in allen erweiternden Komponenten umgesetzt werden. Dies kann zu Fehlern, Inkonsistenzen und zu höherem Aufwand führen, gerade wenn die praktische Anwendung komplexer wird als das überschaubare Layout-Beispiel.

Anders als im vorigen Beispiel (3.4.3) gibt es hier nun zwei Templates die in das Basistemplate gerendert werden sollen. Hierzu muss die `<ng-content>` Direktive weiter konfiguriert werden. Zunächst wird eine Angular-Component in dem Basis-Modul erstellt, in dem die Karte und das Logo vorhanden sind. Zusätzlich kommen nun zwei `<ng-content>` Tags hinzu, die über per `select` an ein bestimmtes übergebenes Template binden.

```
<app-logo [src]="logoSrc"></app-logo>
<app-ol-map [mapUrl]="mapUrl"></app-ol-map>

<div class="right-container
    side-container
    mat-elevation-z8">

  <!-- right transclusion slot -->
  <ng-content select="[right]"></ng-content>

</div>

<div class="left-container
    side-container
    mat-elevation-z8">

  <!-- left transclusion slot -->
  <ng-content select="[left]"></ng-content>

</div>
```

Listing 3.10: base-layout.component.html

Die mapUrl und logoSrc stehen als @Input()-Annotierte Felder in der Komponente zur Verfügung, sie können also von benutzenden Komponenten gesetzt werden:

```
<app-base-layout
  [mapUrl]="mapUrl"
  [logoSrc]="'assets/ecorp.jpg'">

  <app-right right></app-right>

  <app-left left></app-left>

</app-base-layout>
```

Listing 3.11: layout-transclusion.html

Die left und right Attribute sorgen dafür, dass die Komponenten in dem Basis-Modul in die richtigen Container gerendert werden. Sämtliche CSS-Konfiguration liegt hier im Basis-Modul, die erweiternde Anwendung muss sich keinerlei Gedanken um die Positionierung der Elemente machen.

Nun können zusätzlich noch Buttons zum Schließen und Öffnen der Seitencontainer erstellt werden. Diese können dann auch im Basismodul z.B. über dem ng-content platziert werden. Denkbar sind auch Symbole oder einheitliche Überschriften der Container, sofern diese immer die selbe Funktionalität enthalten.

3.4.4 Geschachtelte Transclusion

Zum Umsetzen komplexerer Layouts ist es zudem möglich Transclusion zu schachteln. Es ist also denkbar, dass z.B. im Rechten Container mehrere Slots für Komponenten definiert sind, die dann per Transclusion in das Basislayout und von dort weiter in den Container übergeben werden können.

Die rechte Komponente wird in zwei kleinere Container aufgeteilt die über top-right und bottom-right referenziert werden.

```
<app-base-layout
  [mapUrl]="mapUrl"
  [logoSrc]="'assets/ecorp.jpg'">

    <app-left left></app-left>

    <app-right-top right-top></app-right-top>

    <app-right-bottom right-bottom></app-right-bottom>
</app-base-layout>
```

Listing 3.12: layout-multi-slot-transclusion.html

Hierfür muss das Basislayout so angepasst werden, dass zunächst die beiden neuen Transclusion Slots angelegt und dann per Transclusion in die rechte Basiskomponente weitergereicht werden.

```
<app-logo [src]="logoSrc"></app-logo>
<app-ol-map [mapUrl]="mapUrl"></app-ol-map>

<div class="left-container
    side-container
    mat-elevation-z8">

  <!-- left transclusion slot -->
  <ng-content select="[left]"></ng-content>

```

```
</div>


<div class="right-container
    side-container
    mat-elevation-z8">

  <app-base-right>
    <!-- right top transclusion slot -->
      <ng-content top select="[right-top]"></ng-content>

    <!-- right bottom transclusion slot -->
     <ng-content bottom select="[right-bottom]" right-bottom></ng-content>

  </app-base-right>
</div>
```

Listing 3.13: base-layout.component.html

Das Element wird mit einem Selektor eingefügt und mit einem weiteren an die base-right Komponente weitergereicht, die das Element dann per Multi-Slot-Transclusion an die richtige Stelle in das Template rendert.

```
<mat-card><h1>This is part of the base-right component</h1></mat-card>

<ng-content select="[top]"></ng-content>

<mat-card>
  <h1>This is also the base-right component</h1>
</mat-card>

<ng-content select="[bottom]"></ng-content>

<mat-card>
  And this fancy footer is also part of the base-right component
</mat-card>
```

Listing 3.14: base-right.component.html

So kann über alle erweiternden Anwendungen hinweg eine einheitliche Struktur definiert werden. Das Ergebnis ist in Abbildung 3.7 zu erkennen. Zur Besseren Übersicht wurden die Teile der erweiternden Anwendung hier rot eingefärbt.

Abbildung 3.7: Geschachtelte Transclusion (erweiternde Anwendung rot markiert)

Diese Möglichkeiten von Transclusion lassen sich vielfältig einsetzen um Struktur in GUI-Lastige Anwendungen vorzugeben und HTML-Templates ineinander zu schachteln um Grundstrukturen zu schaffen.

3.5 Extension-Points

In diesem Kapitel wird die Möglichkeit von Extension-Points in der Basiskomponente an einem kleinen Beispiel erläutert. Grundsätzlich soll gezeigt werden, wie eine Basiskomponente Extension Points anbieten kann, die dann von darauf aufsetzenden Komponenten bedient werden können.

3.5.1 Grundprinzip Extension-Points

Der Extension-Point Mechanismus wird oft für die Entwicklung von Plugins verwendet, da hierdurch eine tiefgehende Integration von Funktionalität in die Basiskomponente erreicht werden kann, ohne dass der Code der Basiskomponente hierfür verändert werden muss. Hierfür bietet die Basiskomponente einen Extension-Point an, an dem meist über ein Plugin-Interface eine Erweiterung zu der Anwendung hinzugefügt werden kann.

Dieses Plugin Interface kann beliebig komplex sein und etwa GUI-Elemente zur Darstellung, Lifecycle-Callbacks, bereitgestellte und Konsumierte Services enthal-

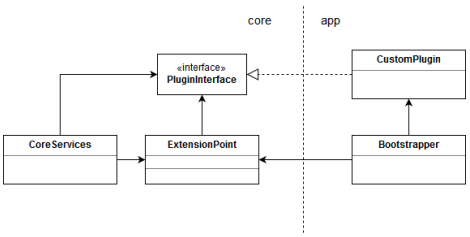

Abbildung 3.8: Mögliche Struktur eines Extension Points

ten. Um die Komponente zu erweitern wird das Plugin-Interface implementiert und die eigene Implementierung dieses Interfaces an den Extension-Point übergeben. Dieser kümmert sich darum, dass die Erweiterung konsistent in die Anwendung integriert wird. Hierfür werden etwaige GUI Elemente an die richte Stelle in Kontextmenüs integriert, Klicks auf Buttons auf die bereitgestellten Methoden im Interface geleitet, sowie die die Lifecycle-Methoden in Kombination mit anderen Komponenten und Plugins verwaltet (Abbildung 3.8).

3.5.2 Beispiel Extension-Points

In einem einfachen Beispiel sollen der Basiskomponente, die die Karte und das Logo anzeigt, zwei Plugins hinzugefügt werden. Alle Plugins sollen am unteren Bildschirmrand über eine Reihe von Buttons aktiviert und deaktiviert werden können und an der Farbe der Buttons soll erkennbar sein, welches Plugin gerade aktiviert ist. Es soll nie mehr als ein Plugin aktiv sein.

Zunächst wird ein möglichst einfaches Plugin-Interface in Form einer abstrakten Klasse deklariert. Das Interface besteht der Einfachheit halber lediglich aus dem Aktivieren bzw. Deaktivieren des Plugins.

```
1  export abstract class MenuButtonPlugin {
2    private _active: boolean;
3
4    protected constructor(public readonly buttonText: string){
5    }
6
7    protected abstract onActiveChanged(active: boolean);
8
9    public get active(): boolean {
10     return this._active;
11   }
```

```
12
13   public set active(value: boolean) {
14     if(value !== this._active) {
15       this._active = value;
16       this.onActiveChanged(value);
17     }
18   }
19 }
```

Listing 3.15: menu-button-plugin.ts

In dem Basismodul wird eine Komponente angelegt, an der die Plugins über einen Service hinzugefügt werden können.

```
1 <app-logo [src]="logoSrc"></app-logo>
2 <app-ol-map [mapUrl]="mapUrl"></app-ol-map>
3
4 <div class="plugin-container">
5   <button mat-raised-button
6           *ngFor="let plugin of extensionPointService.plugins"
7           [color]="plugin.active ? 'primary' : null"
8           (click)="togglePlugin(plugin)">
9     {{plugin.buttonText}}
10  </button>
11 </div>
```

Listing 3.16: extension-point.component.html

Im Code der Komponente zum Template wird in der Methode togglePlugin sichergestellt, dass stets nur ein Plugin aktiv ist.

```
1 public togglePlugin(plugin: MenuButtonPlugin) {
2   if(!plugin.active) {
3   // disable all other plugins
4   this.extensionPointService.plugins
5           .forEach(p => p.active = false);
6   }
7   plugin.active = !plugin.active;
8 }
```

So muss kein Plugin die anderen kennen, sondern bekommt den eigenen Zustand stets von der Anwendung mitgeteilt, ohne Wissen über den eigenen Kontext zu benötigen. Zudem liegt die Kontrolle über den Zustand, die Komposition der Plug-

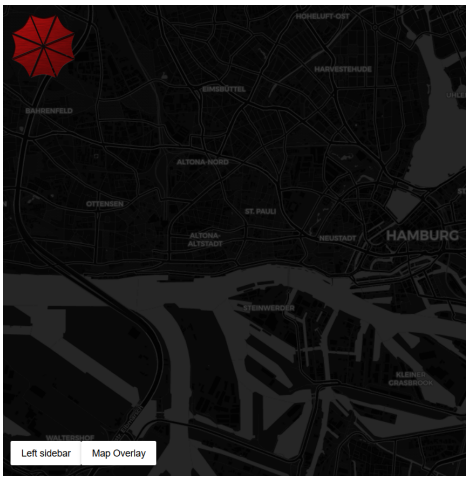

Abbildung 3.9: Anwendung mit Plugin-Buttons

ins, sowie der grafischen Oberfläche der Menü-Leiste, komplett in der Hand der Basiskomponente. Änderungen sind somit auch an einer einzigen Stelle Möglich.

Ohne Plugins zeigt die Komponenten lediglich ein Logo und die Karte an 3.9. Für das Beispiel wurden zwei Plugins hinzugefügt, eins zum Anzeigen eines Sidebar-Menüs 3.10 und eins zum Anzeigen eines Map-Overlays 3.11.

Mit der Konfiguration des Beispiels ist es sogar möglich, zur Laufzeit Plugins hinzuzufügen und zu entfernen, da die Buttons per Data-Binding direkt den Zustand der hinzugefügten Plugins reflektieren. Durch das Erweitern der Plugin-Basisklasse und das Hinzufügen zum ExtensionPointService lassen sich schnell und einfach Funktionalitäten in die Basisanwendung integrieren, ohne dass sich ein Entwickler Gedanken darüber machen muss, wo genau die Menü-Buttons hinzugefügt werden müssen.

Es ist zudem denkbar, die Plugins noch über weitere Schaltflächen anzusprechen, zum Beispiel über ein Drop-Down Menü oder über eine Suche. Um dies zu implementieren kann im Basismodul eine weitere Komponente geschrieben werden, die ebenfalls den ExtensionPointService injiziert bekommt, die Plugins an einer anderen Stelle in die GUI rendert und per Data-Binding an den Zustand bindet. Dadurch dass der Plugin-Zustand über den ExtensionPointService zentral an einer Stelle gehalten wird, können auch mehrere Buttons und Schaltflächen Problemlos mit diesem synchronisiert werden.

All dies kann ohne Wissen der Plugins geschehen, da für die Plugins nur wichtig ist, wie der Zustand gesetzt wird. Weitere Extension Points sind denkbar, um bestimmte

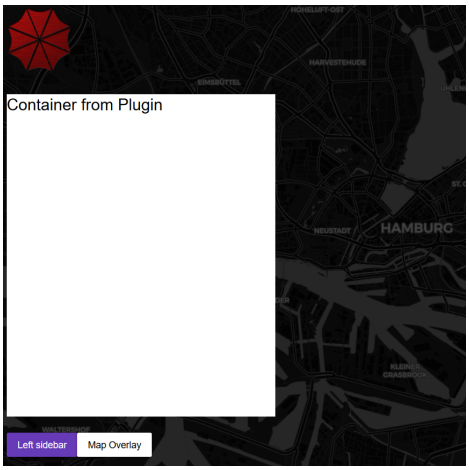

Abbildung 3.10: Anwendung mit aktiviertem Sidebar-Plugin

Abbildung 3.11: Anwendung mit aktiviertem Overlay-Plugin

Werkzeuge nur in bestimmten Kontexten zur Verfügung zu haben oder spezielle Funktionalität zu bieten.

So wäre es zum Beispiel möglich, für die Selektion von Entitäten auf der Karte ein Plugin bereitzustellen, das über einen Button aktiviert/deaktiviert wird, und bei erfolgreicher Selektion die selektierten Entitäten per Event an die Basiskomponente meldet, wo diese weiter verarbeitet bzw. hervorgehoben werden. Dies bietet die Möglichkeit, in der erweiternden Anwendung viele verschiedene und individuelle Werkzeuge zur Selektion anzubieten (Klicken auf der Karte, Envelope aufziehen, Filtern nach Attributen) diese jedoch in eine einheitliche Oberfläche und API der Basiskomponente einzubetten.

Eine weiterer Anwendungsfall könnte sein, nach dem “Convention over Configuration”-Prinzip (vgl. Chen, 2006) bestimmte Plugins in der Basiskomponente bereitzustellen, hierbei aber die Möglichkeit zu bieten, diese Plugins mit eigenen Versionen zu überschreiben. Mit dieser Herangehensweise kann eine Basisanwendung gestaltet werden, die sofort und ohne weitere Konfiguration “Out of the Box” lauffähig ist. Wenn Bedarf zur Anpassung besteht, können diese Anpassungen sukzessive Plugin für Plugin eingefügt werden, wobei die Anwendung während des gesamten Prozesses lauffähig ist.

3.5.3 Extension via Provide-Multi

Neben dem Erstellen eines speziellen Extension-Point Services, ist es auch möglich eine weitere Mechanik der Angular Dependency Injection zu nutzen, um einen Extension-Point Mechanismus zu implementieren. So ist es möglich, für ein Interface mehrere Implementierungen bereitzustellen und sich alle diese Instanzen als Array in benutzende Klassen injizieren zu lassen. Im obigen Beispiel funktioniert dies nicht, da die Plugins Komponenten sind und Komponenten wegen der Bindung an das Template nicht per Dependency Injection injiziert werden können. Denkbar ist dies jedoch immer dann, wenn das Plugin keine eigene Oberfläche mitbringt, sondern als Angular Service implementiert werden kann.

Ein Beispiel aus Angular selbst sind die Http-Interceptoren, mit denen Anwendungsweit Http-Requests modifiziert oder gefiltert werden können, wie aus dem Backend-Bereich bekannt. Um eigene Interceptoren hinzuzufügen, werden die Interceptoren unter dem InjectionToken HTTP_INTERCEPTORS aus dem Http-Modul provided. Die Interceptor Klassen müssen das HttpInterceptor Interface implementieren und können dann über das multi-Attribut parallel provided werden.

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';

@NgModule({
  providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: CustomHttpInterceptor,
      multi: true
    },
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AnotherCustomHttpInterceptor,
      multi: true
    },
  ],
...
})
```

Das obige Beispiel sorgt dafür dass jeder HttpRequest jedes HttpClients durch die Klassen CustomHttpInterceptor und AnotherCustomHttpInterceptor bearbeitet werden. Hierbei können zum Beispiel Tokens oder weitere Header Informationen hinzugefügt werden.

Soll ein solcher Mechanismus implementiert werden, muss zunächst ein Injection-Token und ein Service-Interface in dem Basismodul bereitgestellt werden. In diesem Beispiel soll das Plugin dazu dienen, einen Layer zu der Karte hinzuzufügen. Der Name dient dazu, den Layer später über einen Button ein- und auszublenden.

```
export const MAP_LAYER_TOKEN: InjectionToken<MapLayerService>
  = new InjectionToken<MapLayerService>('MAP_LAYER_TOKEN');

export interface MapLayerService {
  getLayer(): layer.Layer;
  getDisplayName(): string;
}
```

Listing 3.17: map-extension.ts

Um Zugriff auf die bereitgestellten Plugins zu erhalten, wird direkt der Injector von Angular benutzt. Über das Token werden alle Services eingesammelt, die an anderer Stelle provided wurden, als default wird ein leeres Array mitgeliefert, so dass auch

ohne registrierte Plugins ein (in diesem Fall leeres) Ergebnis zurückgegeben werden kann.

```
@Injectable()
export class OlMapService {
  ...

  constructor(private injector: Injector) {
    ...
  }

  _setMap(map: Map) {
    console.log('Map was loaded');
    this.map = map;

    if(!this.layerServices) {
      this.layerServices = this.injector.get(MAP_LAYER_TOKEN, []);
    }

    this.layerServices
      .forEach(service => this.map.addLayer(service.getLayer()));
    this.mapLoaded.emit(this.map);
  }

  ...
}
```

Listing 3.18: map-service.ts

Alles was in der erweiternden Anwendung noch zu tun ist, ist das Interface zu implementieren und über das entsprechende Token mit dem multi-Attribut zu providen.

```
providers: [
  {
    provide: MAP_LAYER_TOKEN,
    useClass: CyclingOverlayService,
    multi: true
  },
  {
    provide: MAP_LAYER_TOKEN,
    useClass: SeamarkOverlayService,
```

```
    multi: true
  }
]
```

Ein dynamisches Hinzufügen und Entfernen der Extensions zur Laufzeit ist mit dieser Methode zwar nicht mehr möglich, dafür lässt sich die Infrastruktur der Angular Dependency Injection nutzen um anwendungsweite und typsichere Extension-Points bereitzustellen. Die Verwaltung der Extensions über einen eigenen Service ist nicht mehr nötig.

Wenn es in der Anwendung mindestens einen Service unter dem InjectionToken gibt, ist es auch möglich den direkten Weg über den Angular-Injector zu vermeiden und sich die Plugins unter Verwendung der Inject-Annotation direkt über den Konstruktor injizieren zu lassen. Dies allerdings funktioniert nur, wenn es mindestens einen Service für das entsprechende InjectorToken gibt.

```
constructor(@Inject(MAP_LAYER_TOKEN) layerServices: MapLayerService[]) {
  ...
}
```

3.6 Fallbeispiel Rails

In diesem Abschnitt soll darauf eingegangen werden, wie einige der in den vorigen Abschnitten genannten Muster in einer realen Anwendung umgesetzt wurden.

Rails ist eine von der Workplace Solutions GmbH entwickelte Software zum georeferenzierten verwalten von Güterwagen mit GPS-Modulen. Auf einer Karte werden die Wagen in Echtzeit dargestellt und können mit verschiedenen Werkzeugen bearbeitet werden. Das Frontend ist eine Single Page Application, die primär für Multi-Touch Tische entwickelt wurde um so kollaboratives Arbeiten an einem Arbeitsplatz optimal zu unterstützen.

Werkzeuge zur Filterung, Visualisierung historischer Daten, Prognosen zur Bestellerfüllung, Bedarfskontrolle sowie Im- und Export in bestehende Systeme oder andere Rails-Instanzen sind ebenfalls Teil der Anwendung oder geplant. Aufgrund der sehr speziellen Fachlichkeiten der Kunden besteht Rails aus einem Anwendungskern mit wiederverwendbaren und konfigurierbaren Basiskomponenten und aus einer für jeden Kunden individuell entwickelten Anwendung, die auf diesem Kern aufsetzt.

Die Anwendung ist zur Zeit dieser Arbeit noch in einem recht frühen Stadium der Entwicklung, wird aber von mehreren Kunden schon parallel zu anderen produktiven Systemen eingesetzt.

3.6.1 Anwendungsarchitektur

Rails besteht aus einem Java Spring Boot Backend und einem Angular Frontend und ist als Single Page Application konzipiert. Da mit den Wagenpositionen auch viele Echtzeitdaten verarbeitet werden, bietet das Backend neben einer ReST-API auch Change-Events per Websockets über das STOMP Protokoll an. Ein Client kann so per HTTP-GET initial den aktuellen Zustand (z.B. der Wagendaten) laden, und sich per Websockets über alle Änderungen einzelner Wagen und Positionsdaten informieren lassen.

Bestehenden Systeme der Kunden werden über eine auf den Kunden zugeschnittene Schnittstellenkomponente mit entsprechender Konvertierungslogik angebunden. Die Entitäten werden auf das interne Datenmodell gemapped und an Services delegiert, die die Persistenz via Repositories übernehmen, und Change-Events bei Änderungen an den Entitäten verteilen. ReST-Controller bieten per HTTP Zugriff auf die Entitäten und leiten die Change-Events mit dem STOMP-Protokoll via Websocket weiter. Ein Tokenbasiertes Security Konzept ist ebenfalls im Backend implementiert und sichert den Zugriff sowohl auf die HTTP-Schnittstellen als auch auf die Websockets.

Im Client ist der Hauptteil der Anwendung eine Karte, die mit dem Openlayers Framework umgesetzt wurde. Eine Filterbank, eine Merkliste mit Im- und Export Funktionen, eine Suche nach Wagennummern und weitere Werkzeuge werden am Bildschirmrand über der Karte angeordnet und können je nach Bedarf ein und ausgeklappt werden. Durch die Kombinationen dieser möglichst einfach gehaltenen Werkzeuge, sollen dem Anwender umfassende Möglichkeiten zur räumlichen Interaktion mit den Daten geboten werden (siehe Abbildung 3.12).

3.6.2 Verwendete Konzepte

Da die Fachlichkeit und die Anforderungen der einzelnen Kunden stark auseinandergehen, muss das Datenmodell außerhalb der Basis in der jeweiligen Anwendung für den Kunden umgesetzt werden. Dies gilt sowohl für den Client-Teil als auch für den Server-Teil von Rails. Die Herausforderung besteht darin, trotz der sehr

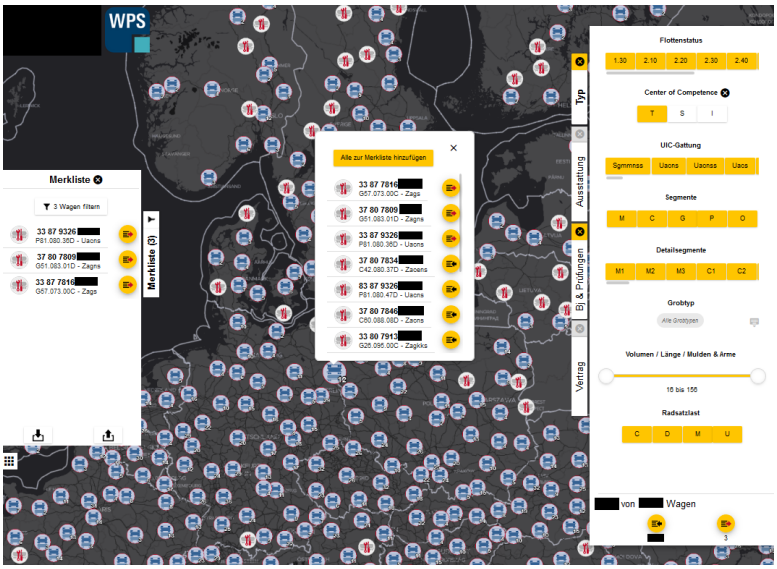

Abbildung 3.12: Screenshot Rails Anwendung

unterschiedlichen Fachlichkeiten möglichst viele Gemeinsamkeinten zu finden, die in die Basisanwendung abstrahiert werden können.

Serverseitig sind das vor allem abstrakte generische Basisservices, die z.B. das Handling der Change-Events und das Speichern in die JPA-Repositories von Spring übernehmen, das Cachen von Hintergrundkarten, und das Rendern statischer Features basierend auf einem Shapefiles. Da sich sehr viel Logik um das Mapping des speziellen fachlichen Datenmodells dreht, konnte im Server vergleichsweise wenig Code in die Basis ausgelagert werden.

Clientseitig gab es deutlich mehr Möglichkeiten. So konnten Basisservices für die Synchronisation mit dem Server via HTTP und STOMP, die Synchronisation eines solchen Services mit der Karte, und viele GUI Bausteine in dem Basisteil der Anwendung implementiert werden. Beispielhaft sind hier eine Popup-Komponente auf der Karte (Abbildung 3.13), und drei Filterwerkzeuge (Abbildungen 3.14, 3.15 und 3.16) als Screenshots zu sehen, die in der Basis implementiert und von der eigentlichen Anwendung lediglich konfiguriert und benutzt werden.

Da die Entwicklung der Anwendung mit Angular Version 5 begonnen hat, ist im Client die Aufteilung in Library und Applikation noch nicht abgeschlossen. Es gibt jedoch eine Modulstruktur, die die Aufteilung gut unterstützt. Nach einem Upgrade

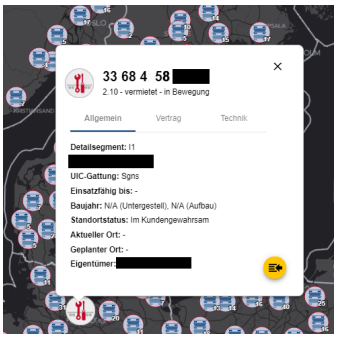

Abbildung 3.13: Karten-Popup mit Detailinformationen eines Wagens

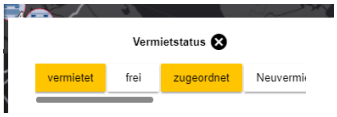

Abbildung 3.14: Listenbasierter Attributfilter

Abbildung 3.15: Numerischer Range-Filter

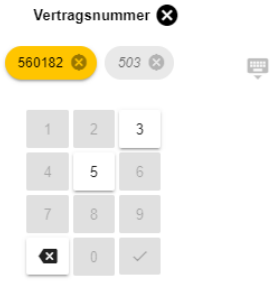

Abbildung 3.16: Filter für kurze IDs und Nummern

auf Angular 6 lässt sich diese Aufteilung mit der Deklaration der Modul-API und einem Anpassen der Imports relativ einfach umsetzen.

Bei der zukünftigen Entwicklung weiterer Rails-Anwendungen für weitere Kunden und dem Vernetzen dieser Anwendungen untereinander, ist davon auszugehen, dass sowohl im Server als auch im Client noch weitere Aspekte in die Basis von Rails ausgelagert werden können. Dies wird im Laufe der Entwicklung dafür sorgen, dass Rails zu einem Anwendungsframework wird, das unter anderem unter Berücksichtigung der in dieser Arbeit genannten Konzepte, die Entwicklung weiterer Kundenanwendungen beschleunigt und auf eine gemeinsame Basis stellt.

4 Zusammenfassung und Ausblick

In diesem Abschnitt soll noch einmal die Arbeit zusammengefasst und ein Ausblick auf mögliche weitere Inhalte gegeben werden, die es nicht in diese Arbeit geschafft haben.

4.1 Zusammenfassung

In dieser Arbeit wurden zunächst im Grundlagen Kapitel 2 das Konzept von Single Page Applications und einige grundlegende Konzepte des Angular Frameworks beleuchtet. Hierbei wurde deutlich, wieso Single Page Applications gut funktionieren und architektonisch sauberer strukturiert werden können, als herkömmliche Webanwendungen. Zudem wurde erläutert, wieso die hohe clientseitige Komplexität einer solchen Anwendung nach einem Framework wie Angular verlangt, um die Herausforderungen des dynamischen Renderns auf einer einzigen HTML-Seite zu meistern. Bei den Angular Grundlagen wurden die Vorteile von Typescript und die Konzepte von Komponenten, Dependency Injection und Modulen anhand einiger Beispiele dargestellt um einen leichteren Einstieg in die folgenden Kapitel zu ermöglichen. Zusätzlich wurde das RxJs Framework kurz erläutert, da Angular stark darauf aufsetzt.

Im nächsten Kapitel 3 wurden diese Konzepte an Beispielen angewandt, um eine Strukturierung in eine Basis und darauf aufsetzende Anwendungen zu ermöglichen. Zunächst ging es hier um das Setup eines Multi-App Projektes mit dem Angular-CLI, sowohl in der neuen Version 6 als auch in der alten Version davor. Damit verbunden ist auch eine Erläuterung des Library-Supports ab Version 6, der die Möglichkeit bereitstellt direkt über das Angular-CLI Libraries für npm zu bauen. In den weiteren Abschnitten wurden dann anhand einiger Codebeispiele Möglichkeiten von Basiskomponenten, providen spezieller Serviceausprägungen, Transclusion und Extension Points erläutert.

So wurde gezeigt, wie ein Basisservice zum Handling von Entitäten bereitgestellt werden kann, ohne den Typ der Entitäten zu kennen und wie in der Basisanwendung Service Interfaces und mit der Angular Dependency Injection die speziellen Ausprägungen bereitgestellt werden können.

Transclusion wurde als Werkzeug für wiederverwendbaren Template Code und zur Vorgabe einer GUI Grundstruktur in einer Basisanwendung vorgestellt. Hierbei wurde außerdem das Prinzip geschachtelter Transclusion anhand einiger Codebeispiele und Screenshots verdeutlicht.

Außerdem wurden mehrere Möglichkeiten für das Erstellen von Extension Points (dynamisch zur Laufzeit und Statisch über die Angular Dependency Injection) anhand einiger Beispiele erläutert.

Zu guter letzt wurde mit Rails ein Fallbeispiel vorgestellt, an dem viele der genannten Konzepte bereits in einer produktiv entwickelten Anwendung umgesetzt wurden um neben kleinen und abstrakten Beispielen auch ein Beispiel aus einem realen Projekt zu vorzustellen und so zu zeigen, dass die Konzepte durchaus auch im realen Entwicklungsbetrieb umsetzbar sind.

Zusammenfassend lässt sich sagen, dass Angular vielfältige Lösungen und Möglichkeiten bietet, um solide und erweiterbare Basisstrukturen zu schaffen und eine Codebasis in mehreren Anwendungen zu nutzen.

4.2 Ausblick

Obwohl diese Arbeit einige Konzepte und Möglichkeiten abgedeckt hat, gibt es noch einige Aspekte die im Zuge einer weiteren umfangreicheren Arbeit genauer beleuchtet werden könnten.

Das Angular Framework wird ständig weiterentwickelt. Halbjährliche Releases bringen Neuerungen, die dazu führen können, dass in dieser Arbeit vorgestellte Beispiele auf elegantere Art und Weise umsetzbar sind. Gerade bei dem recht neuen Library Support des Angular CLI dürften in nächster Zeit einige neue Möglichkeiten entstehen, mit denen die Beispiele aus dieser Arbeit sinnvoll ergänzt werden könnten.

Eine interessante Erweiterung wäre auch das Strukturieren einer größeren Anwendung mit einer konsequenten Extension-Point Architektur. Die Anwendung würde nur einen Rahmen für verschiedene Werkzeuge bieten, die dann als Plugins über einen Extension-Point Mechanismus bereitgestellt werden. So würden die Schaltflä-

chen der Plugins in die jeweiligen Toolbars, Kontextmenüs und Funktionssuchen der Basisanwendung integriert und wären auch zur Laufzeit austauschbar. Eine solche Architektur würde viele interessante Herausforderungen mit sich bringen, um Abhängigkeiten und Kommunikation zwischen Plugins und eine stimmige Oberfläche umzusetzen.

Im aktuellen Entwicklungsstand der Rails Anwendung sind einige der in dieser Arbeit behandelten Konzepte noch nicht in letzter Konsequenz umgesetzt. Mit fortschreitender Entwicklung dürfte Rails früher oder später einen Stand erreichen, der so viele für eine Arbeit dieser Art relevante Aspekte abdeckt, dass die Anwendung als Basis für eine eigene Arbeit dienen kann. Statt einfacher prototypischer Beispiele könnten dann durchgängig reale Beispiele aus der Anwendung dienen. Im Entwicklungsbetrieb entstehende Probleme können genutzt werden, um die Konzepte noch weiter verfeinern und weitere Muster und Konzepte hinzuziehen, die sich im Projektalltag bewährt haben.

Insgesamt hat diese Arbeit Konzepte und Beispiele vorgestellt, die anhand einer umfangreicheren Anwendung erprobt und erläutert werden könnten. Die für Webframeworks übliche schnelle Weiterentwicklung des Angular Frameworks und weitere Anwendungsbereiche bieten vielfältige Möglichkeiten, um sich in noch weiterer Tiefe mit dem Themenbereich dieser Arbeit auseinanderzusetzen.

Literaturverzeichnis

[angular] angular: *npm*. – URL https://angular.io/. – Zugriffsdatum: 2018-06-11

[Angular-Github 2017] Angular-Github: *Asset Configuration*. 2017. – URL https://github.com/angular/angular-cli/wiki/stories-asset-configuration. – Zugriffsdatum: 2018-04-21

[Angular-Github 2018] Angular-Github: *Multiple Apps Integration*. 2018. – URL https://github.com/angular/angular-cli/wiki/stories-multiple-apps. – Zugriffsdatum: 2018-04-21

[Chen 2006] Chen, Nicolas: *Convention over Configuration*. 2006. – URL http://softwareengineering.vazexqi.com/files/pattern.html. – Zugriffsdatum: 2018-05-21

[Fischer und Hanenberg 2015] Fischer, Lars ; Hanenberg, Stefan: An Empirical Investigation of the Effects of Type Systems and Code Completion on API Usability Using TypeScript and JavaScript in MS Visual Studio. In: *SIGPLAN Not.* 51 (2015), Oktober, Nr. 2, S. 154–167. – URL http://doi.acm.org/10.1145/2936313.2816720. – ISSN 0362-1340

[Gamma u. a. 1995] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0-201-63361-2

[Hanenberg 2010] Hanenberg, Stefan: Doubts About the Positive Impact of Static Type Systems on Programming Tasks in Single Developer Projects - an Empirical Study. In: *Proceedings of the 24th European Conference on Object-oriented Programming*. Berlin, Heidelberg : Springer-Verlag, 2010 (ECOOP'10), S. 300–303. – URL http://dl.acm.org/citation.cfm?id=1883978.1883998. – ISBN 3-642-14106-4, 978-3-642-14106-5

[Kuuskeri 2011] Kuuskeri, Janne: Experiences on a Design Approach for Interactive Web Applications. In: *Proceedings of the 2Nd USENIX Conference on Web Application Development*. Berkeley, CA, USA : USENIX Association, 2011 (WebApps'11), S. 8–8. – URL http://dl.acm.org/citation.cfm?id=2002168.2002176

[Larsen 2018] Larsen, Hans: *Library support in Angular CLI 6*. 2018. – URL https://github.com/angular/angular-cli/wiki/stories-create-library. – Zugriffsdatum: 2018-05-21

[Lilienthal 2017] Lilienthal, Carola: *Langlebige Software-Architekturen - Technische Schulden analysieren, begrenzen und abbauen*. Heidelberg : dpunkt.verlag, 2017. – ISBN 978-3-960-88245-9

[Malcher u. a. 2017] Malcher, Ferdinand ; Woiwode, Gregor ; Hoppe, Johannes ; Koppenhagen, Danny: *Angular - Grundlagen, fortgeschrittene Techniken und Best Practices mit TypeScript - ab Angular 4, inklusive NativeScript und Redux*. Heidelberg : dpunkt.verlag, 2017. – ISBN 978-3-960-88206-0

[npm 2018] npm: *npm*. 2018. – URL https://www.npmjs.com/. – Zugriffsdatum: 2018-06-14

[Petersen u. a. 2014] Petersen, Pujan ; Hanenberg, Stefan ; Robbes, Romain: An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse. In: *Proceedings of the 22Nd International Conference on Program Comprehension*. New York, NY, USA : ACM, 2014 (ICPC 2014), S. 212–222. – URL http://doi.acm.org/10.1145/2597008.2597152. – ISBN 978-1-4503-2879-1

[RxJs-Github 2018] RxJs-Github: *RxJS 5*. 2018. – URL https://github.com/ReactiveX/rxjs/tree/stable. – Zugriffsdatum: 2018-06-09

[spring 2018] spring: *Spring*. 2018. – URL https://spring.io/. – Zugriffsdatum: 2018-06-14

[Stackoverflow 2018] Stackoverflow: *Stackoverflow Developer Survey Results 2018*. 2018. – URL https://insights.stackoverflow.com/survey/2018/. – Zugriffsdatum: 2018-06-04

[Starke 2015] Starke, Gernot: *Effektive Softwarearchitekturen - Ein praktischer Leitfaden*. M : Carl Hanser Verlag GmbH Co KG, 2015. – ISBN 978-3-446-44406-5

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 27. Juli 2018 Niklas Kopp