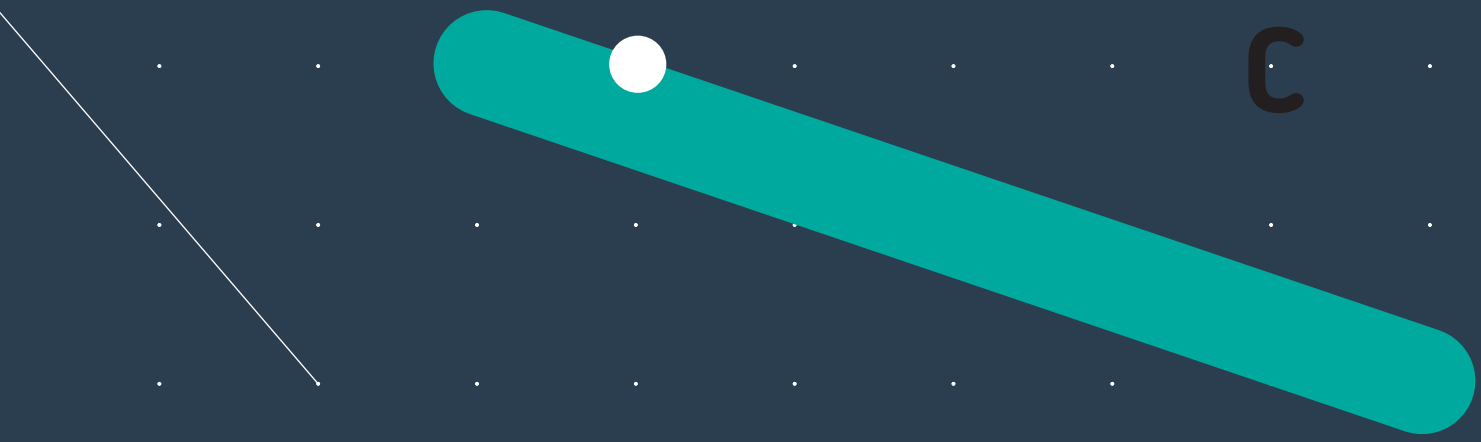
Hochschule Konstanz
Fakultät Informatik

Autor
Jonas Reinwald

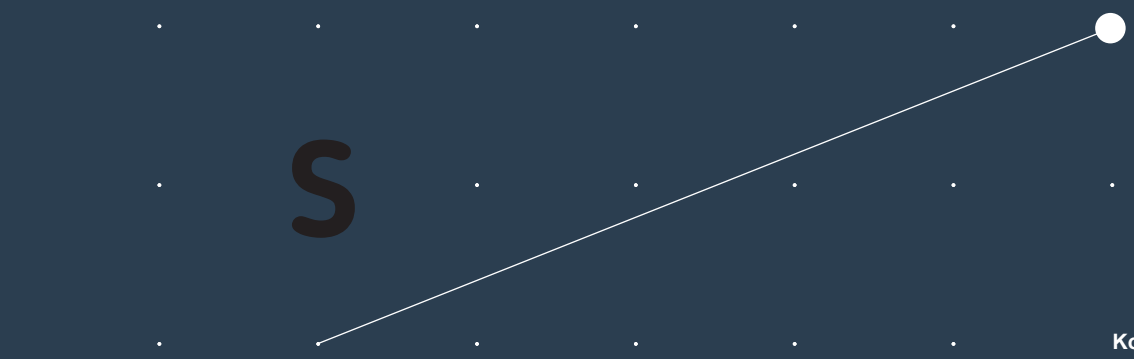
Immatrikulationsnummer
293320

E-Mail-Adresse
jonas.reinwald@htwg-konstanz.de



Bachelorthesis

Konzipierung der Architektur einer Darstellungsschicht basierend auf aktuellen Webtechnologien im Hinblick auf eine zukünftige Anbindung an das Programm combit Relationship Manager



Konstanz, 25. Februar 2019

Hochschule Konstanz
Fakultät Informatik

Bachelorthesis

Konzipierung der Architektur einer Darstellungsschicht basierend auf aktuellen Webtechnologien im Hinblick auf eine zukünftige Anbindung an das Programm combit Relationship Manager

von

Jonas Reinwald

Zur Erlangung des akademischen Grades eines

Bachelor of Science (B.Sc.)

an der Hochschule Konstanz für Technik, Wirtschaft und Gestaltung

Vorgelegt von:	Jonas Reinwald Wessenbergstr. 13 78462 Konstanz 293320 (Matr.-Nr.)
Erstbetreuer:	**Prof. Dr. Marko Boger**
Zweitbetreuer:	**Dipl.-Inf. Alexander Horak** combit GmbH Untere Laube 30 78462 Konstanz
Ausgegeben am:	01.12.2018
Eingereicht am:	28.02.2019

Abstract

Webapplikationen in Form von mobilen Apps, Single-Page-Applikationen, Streaming-Seiten und vielen weiteren Arten machen heute bereits einen sehr großen Teil häufig genutzter Software aus. Laut StackOverflow sind die meisten Softwareentwickler in weborientierten Firmen angestellt [2018a] und arbeiten respektive mit entsprechenden Technologien [2018b]. In Zukunft wird der Anteil webbasierter Software im Vergleich zu traditioneller Software vermutlich noch weiter wachsen, vor allem im Bereich deutscher CRM-Software nimmt die Bedeutung des mobilen Zugangs im Vergleich zum Vorjahr deutlich zu [vgl. Bahr und Capterra 2019]. Auch für Unternehmen ist es langfristig unabdingbar, diesem Trend zu folgen, um Kunden flexiblere Softwarelösungen zur Verfügung zu stellen und nicht den Eindruck zu erwecken, es könne nicht mit modernen Entwicklungen Schritt gehalten werden.

Die Firma *combit GmbH* entwickelt mit dem *combit Relationship Manager (cRM)* eine Software für das Management von Kundenbeziehungen (Customer Relationship Management (CRM)). Diese besteht zum gegenwärtigen Zeitpunkt aus einer Desktopapplikation, einer Webapplikation und einer mobilen Webapplikation, welche unabhängig voneinander entwickelt sowie benutzt werden. Im Rahmen dieser Bachelorthesis soll ein Architekturkonzept für eine einheitliche Oberfläche (ThinClient[1]) basierend auf moderner Webtechnologie erarbeitet werden, mithilfe derer die drei aktuellen Ausführungen des *cRM* langfristig abgelöst werden können.

Zur Erstellung dieses Konzepts beschäftigt sich diese Arbeit unter anderem mit der Evaluierung eines geeigneten Frontend-Frameworks bzw. einer Frontend-Bibliothek, der Kommunikation zwischen neuem UI-Layer und Backend sowie der Unterstützung aller bisherigen Features der Desktopapplikation. Zwei besonders spannende Herausforderungen ergeben sich aus dem Anspruch der Beibehaltung der Flexibilität der momentanen Oberfläche, welche von jedem Nutzer[2] individuell anpassbar ist. Diese Flexibilität soll durch eine automatische Konvertierung zu einem von der neuen UI darstellbaren und weiterhin anpassbaren Format erhalten bleiben. Zum anderen ist es aber auch wichtig, webspezifische Anforderungen, vor allem ein responsives Layout und Design, mit der Flexibilität der Oberfläche zu vereinen.

[1] Clientapplikation, bei welcher sich der Großteil der Logik auf einem externen Server befindet

[2] Aus Gründen der leichteren Lesbarkeit wird in der vorliegenden wissenschaftlichen Arbeit das generische Maskulinum verwendet, welches nicht geschlechterspezifisch gilt, sondern Personen aller Geschlechter mit einschließt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

Bezeichnung	Beschreibung	Seiten
CLI	Command-Line-Interface	23, 45, 53
COM	Component Object Model	11, 12
CRA	create-react-app	45
CRM	Customer Relationship Management	iii, 5
cRM	combit Relationship Manager	iii, v, ix, 2, 5–9, 11–13, 15, 18, 19, 21, 39, 49, 57, 61
DSGVO	Datenschutz-Grundverordnung	5, 17
LL	List & Label	6, 10
SSR	Server-Side-Rendering	vi, 29
WYSIWYG	What You See Is What You Get	10

1

Einleitung

Dieses Kapitel soll in die Thematik ein einführen, die Motivation für die Arbeit erläutern und einen Überblick über im Voraus antizipierte Schwierigkeiten bezüglich der Umsetzung des Architekturkonzepts geben.

1.1. Vorgehen und Gliederung

In Kapitel 2 werden Anforderungen (unter anderem anhand bisheriger Features) ermittelt. Darauf aufbauend werden in Kapitel 3 die in Betracht gezogenen Technologien und deren Zusammenspiel untereinander vorgestellt. Dazu gehören die zu nutzende Programmierumgebung (Sprache, Testframework, kontinuierliche Integration[1]), der Vergleich verschiedener Frontend-Frameworks, die anhand der in Kapitel 2 formulierten Anforderungen bewertet werden und die Protokolle zur Kommunikation mit dem Backend. Anschließend wird die Ausarbeitung des Konzepts mit den in Kapitel 3 ausgesuchten Technologien und der Aufbau der API zur Kommunikation mit dem Backend in Kapitel 4 beschrieben. Kapitel 5 stellt den Entwicklungsprozess der beispielhaften Umsetzung des erstellten Konzepts dar und in Kapitel 6 wird das Ergebnis kritisch hinterfragt, die tatsächlich aufgetretenen Schwierigkeiten mit den zuvor Erwarteten verglichen und ein Ausblick auf weitere Entwicklungsschritte gegeben.

[1] CI: Continuous Integration

1.2. Motivation

Ein Großteil der heute genutzten Software kann über eine Weboberfläche angesprochen werden. Dies hat den Vorteil, dass keine langwierigen und komplizierten Installationen vorgenommen werden müssen und, eine bestehende Internetverbindung vorausgesetzt, die Applikation von überall erreichbar ist. Die Firma *combit GmbH* hat diese Entwicklung früh erkannt und bietet für den *cRM* eine entsprechende Weboberfläche an. Diese existiert bereits seit vielen Jahren und basiert, bedingt durch den damaligen Stand der Technik, auf nicht mehr zeitgemäßen Technologien. Da der Fokus der Entwicklung — bestimmt durch die Bedürfnisse der Kunden — auf der Desktopanwendung liegt, waren nicht mehr genügend personelle Ressourcen für den allmählichen Ersatz der Technologien durch moderne Alternativen vorhanden.

Webtechnologien bringen nicht nur Vorteile für ihre Nutzer mit sich, sondern beinhalten auch einige interessante Aspekte für das Entwicklerteam selbst. So kann dieses moderne Projekte und Konzepte, welche durch die vielen technischen Fortschritte in diesem Bereich entstanden sind, kennenlernen und hat eine breite Auswahl an bestehenden Projekten für eine solide Basis des eigenen Produkts. Ein Beispiel für einen solchen Fortschritt sind Programme, die es erlauben, Webseiten wie normale Programme nativ auf einem Desktopcomputer darstellen zu können. Das Ziel hierbei ist, langfristig nur noch eine einzige Version für alle Umgebungen entwickeln zu müssen, wodurch Ressourcen gespart werden können. Durch den einheitlichen Fokus, so die Erwartung, soll zudem ein besseres Produkt entstehen. Dieses Potential der Webtechnologien wird dadurch noch weiter gesteigert, dass durch einfache Sprachen wie *HTML*, *CSS* und *JavaScript*, welche nicht kompiliert werden müssen und auf der Mehrheit der modernen Geräte lauffähig sind, schnell viele Iterationen der Software erstellt werden können. Anhand der genannten Gründe wurde entschieden, das Thema Web mit einer innovativen Herangehensweise, einer veränderten Perspektive und modernen Technologien neu anzugehen, um den *cRM* sowohl extern als auch intern noch attraktiver zu gestalten.

1.3. Erwartete Schwierigkeiten

Bereits im Vorfeld soll ein kleiner Überblick über die Aspekte dieser Arbeit gegeben werden, welche im Laufe der Umsetzung zu Schwierigkeiten führen könnten. Die vorherige Auseinandersetzung mit eventuell auftretenden Problemen ermöglicht die Antizipation ebendieser und somit eine vorausschauendere Auseinandersetzung mit der Aufgabe. Zudem dient sie der Reflexion der eigenen Fähigkeiten und damit deren Ausbau.

Weitere Informationen zu den hier aufgelisteten Punkten werden in Kapitel 2 aufgeführt.

Eine der größten Herausforderungen wird sein, eine Umsetzung des User Interface zu finden, welche auch weiterhin ebenso oder zumindest annähernd anpassbar ist, wie dies in der aktuellen Version der Fall ist. Da das Layout dateibasiert gespeichert wird und im Browser nicht auf diese Dateien zugegriffen werden kann, muss ein neues Konzept für die Persistenz erarbeitet werden, mithilfe dessen die Verteilung über das Netzwerk einfacher möglich ist. Neben dem angesprochenen Layout gibt es auch noch weitere Informationen, die momentan nur dateibasiert auf dem jeweiligen Client gespeichert sind. Diese müssen ebenfalls alle in ein neues Konzept überführt werden.

Um auf alle diese Informationen sowie die eigentlichen Daten, die dem Nutzer angezeigt werden sollen, zugreifen zu können, wird eine entsprechende Schnittstelle (API) benötigt. Das Design dieser API ist kritisch, da sich nachträglich nötige Änderungen auf jede Komponente auswirken, was wiederum Anpassungen an ebendiesen erfordert.

Weiterhin gilt es zu entscheiden, welche Features im Web überhaupt nicht oder nur mit zu vielen Schwierigkeiten, etwa eine auf Desktopumgebungen ausgelegte Scripting-Unterstützung, oder die Interaktion und Kommunikation mit externen Prozessen, umsetzbar sind. Diese Punkte sind bei der Erstellung der Anforderungen entsprechend zu beachten. Hier muss abgewogen werden, inwiefern die Einschränkungen den Nutzer limitieren könnten und die Attraktivität der neuen Oberfläche verringern.

2

Anforderungsanalyse

Um zu wissen, welche Features und Eigenschaften im neuen Konzept beachtet werden müssen, werden in diesem Kapitel vor der Erstellung zunächst die Anforderungen ermittelt.

2.1. Ausgangszustand und Produktüberblick

Zum tiefgreifenderen Verständnis der Anforderungen an das zu entwickelnde Architekturkonzept, muss zunächst einmal der Ausgangszustand, das heißt der jetzige Desktopclient des *cRM* betrachtet werden. Wie am Begriff CRM bereits zu erkennen, handelt es sich dabei um ein Programm, das der Verwaltung und Vernetzung von Kunden- und Kontaktdaten dient. Neben den typischen Einsatzbereichen von CRM-Produkten, wie die Verwaltung von Kundendaten, Terminen und Aufgaben, legt die *combit GmbH* allerdings besonderen Wert darauf das Programm möglichst flexibel zu gestalten, um dadurch zahlreiche Kundengruppen mit ihren spezifischen Anforderungen an die Einsatzbereiche der Software zu bedienen. Einige Beispiele hierfür sind die mitgelieferten, fertigen Lösungen (im Kontext des *cRM* „Solutions“ genannt), mit denen unter anderem Immobilien gehandhabt, die Vermittlung von Arbeitsstellen an Drittfirmen organisiert oder ein Ticket-System aufgebaut werden können. Zudem ist eine Musterlösung für den Datenschutz-Grundverordnung (DSGVO)-konformen Umgang mit sämtlichen Daten integriert. Unabhängig von diesen Lösungen stehen bestimmte Funktionalitäten durchweg zur Verfügung: Import und Export von Daten in viele verschiedene Formate, Versand von (Serien-) E-Mails oder etwa die Generierung von Berichten (Reports)

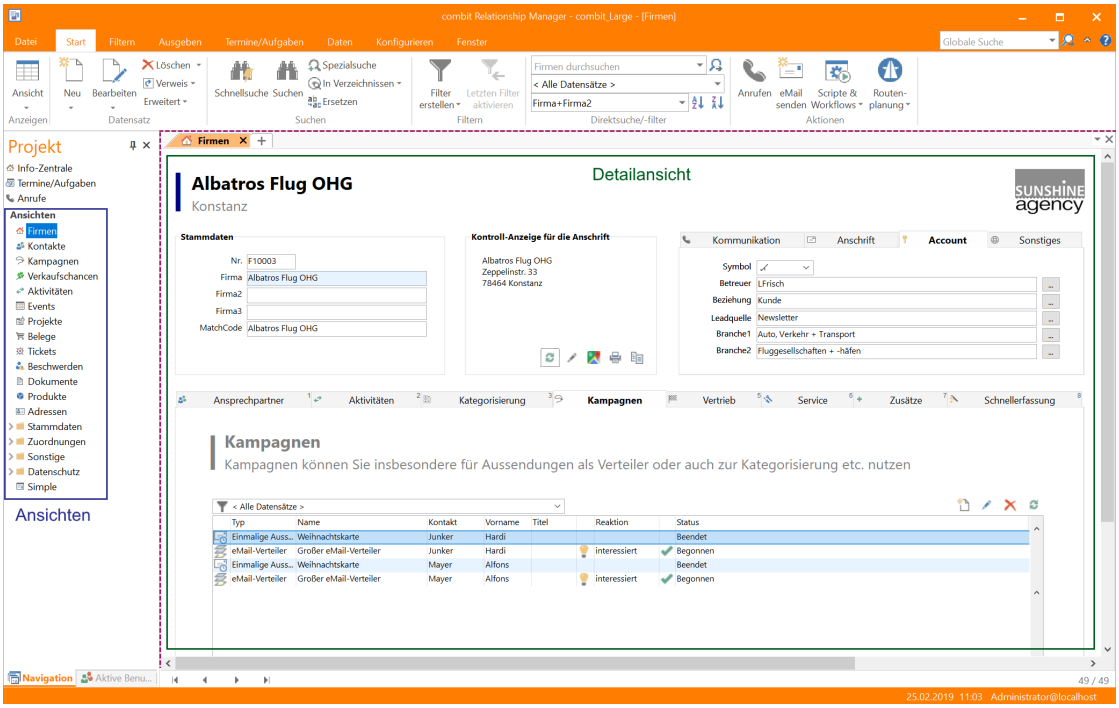


Abbildung 2.1: Grafische Oberfläche des *cRM* Desktopclient (Februar 2019)

durch ein zweites internes Produkt „*List & Label (LL)*“. Diese bauen jedoch auf den Daten einer Lösung auf. Eine Solution stellt die grundsätzliche Organisation der Daten in bestimmte Bereiche oder Szenarien dar und kann von jeder Firma individuell konfiguriert werden. Jede Solution ist durch eine Menge von Datenbanktabellen und darauf basierenden Ansichten zur Visualisierung der in den Tabellen enthaltenen Entitäten und Relationen definiert. In der in Abbildung 2.1 gezeigten Standard-Solution „Large“ ist links eine Liste der verfügbaren Ansichten und rechts die Detailansicht einer Entität der „Firmen“-Ansicht zu sehen. Außerdem wird hier auch die allgemeine Trennung zwischen Haupt-UI und ansichtsspezifischer UI ersichtlich. Anstelle der Detailansicht kann auch eine Listenansicht (tabellenförmige Auflistung aller Entitäten), eine Berichtsansicht, eine Webansicht und eine Kombinationsansicht, bei der sowohl die Übersichtsliste als auch die Details sichtbar sind, angezeigt werden.

Aus technischer Sicht ist der *cRM* in drei Schichten getrennt. Die zu verwaltenden Daten werden entweder in einer *Microsoft SQL Server*- oder einer *PostgreSQL*-Datenbank gespeichert. Eine Besonderheit hierbei ist die Tatsache, dass Relationen zwischen Datenbanktabellen nicht durch Fremdschlüssel und zugehörige Relationen auf Datenbankebene, sondern durch virtuelle Relationen zwischen einzelnen Ansichten im *C++*-Kern der Anwendung, realisiert sind. Dies erlaubt einen flexibleren Umgang mit diesen und vereinfacht das Ändern von Datenbanktabellen, Ansichten und Relationen direkt aus der Oberfläche des *cRM*. Der *cRM*-Core bildet die zweite Schicht und ist für

den Datenbankzugriff und die Business Logic[1] verantwortlich. Diese beiden Schichten sollen von der Erstellung der neuen Weboberfläche möglichst unberührt bleiben. Einzig die API-Anbindung für die UI muss an dieser Stelle integriert werden — betrachtet wird die API in dieser Arbeit jedoch nur aus Sicht der Oberfläche, die Integration im Backend findet zu einem späteren Zeitpunkt statt. Aufbauend auf dem Kern existieren parallel der Desktopclient, ebenfalls in *C++* geschrieben, der WebAccess für Desktopbrowser und der WebAccess Mobile für mobile Endgeräte. Zwischen Core und Webaccess (Mobile) gibt es noch einen mit *ASP.NET*[2] realisierten *.NET*-Wrapper, welcher den Zugriff auf Daten per HTTP ermöglicht.

Abbildung 2.2 zeigt eine Übersicht der momentanen Architektur, Abbildung 2.3 hingegen, wie die Architektur in Zukunft aufgebaut sein könnte.

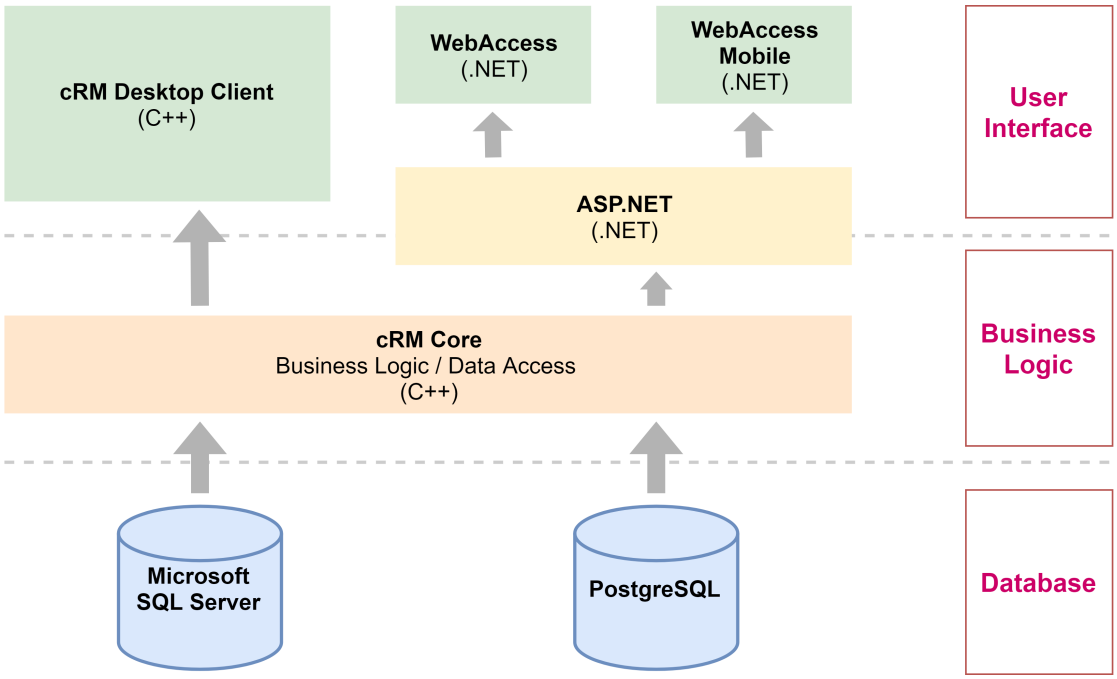


Abbildung 2.2: Aktueller technischer Aufbau des *cRM*

[1] Anwendungslogik zwischen UI und Datenbank, welche programmspezifische (Verarbeitungs-) Regeln enthält

[2] Auf dem *.NET*-Framework aufbauende Tools und Bibliotheken zum Erstellen von Web-Apps

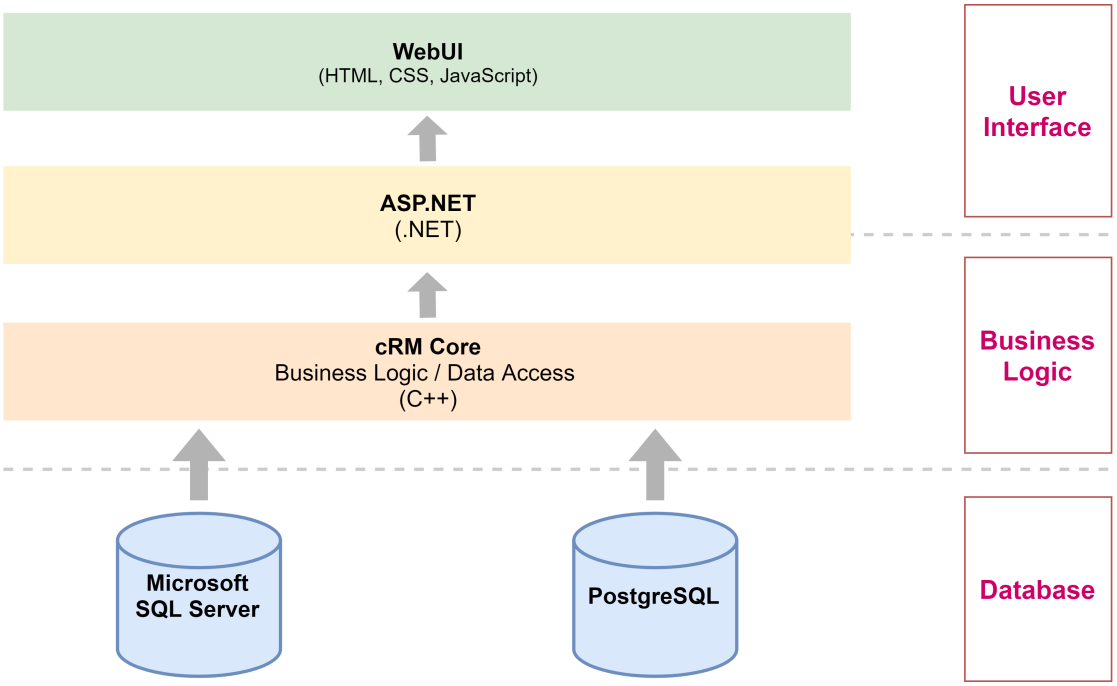


Abbildung 2.3: Möglicher, zukünftiger technischer Aufbau des *cRM*

2.2. Funktionale Anforderungen des combit Relationship Manager

Dieser Teil des Kapitels befasst sich mit den funktionalen Anforderungen, welche sich direkt aus dem *cRM* und dessen bisherigen Eigenschaften und Funktionen ergeben.

2.2.1. Automatisierte Erstellung der neuen UI

Die meisten Kunden haben entweder bereits ihre eigenen Ansichten erstellt oder benutzen angepasste Versionen der mitgelieferten Designs. Wird als langfristiges Ziel definiert, die komplette UI-Schicht ins Web zu überführen, muss das Ergebnis dieses Prozesses für die Benutzer leicht und zugänglich genug sein, um nicht schon im Voraus abgelehnt zu werden. Es ist daher essenziell, dass sämtliche bisherigen Ansichten mit minimalem Aufwand seitens der Nutzer im Web weiterverwendet werden können. Der minimale Aufwand meint dabei, dass im Ergebnis lediglich kleine Anpassungen tolerierbar sind, jedoch nicht die Neuerstellung jeder einzelnen Ansicht. Eines der Hauptziele dieser Arbeit ist es daher, eine automatische Umwandlung bestehender Ansichten zu ermöglichen. Hierfür muss die aktuelle UI-Spezifikation analysiert und eine neue Spezifikation erstellt werden, mithilfe derer Implementierungen aus den vorherigen abgeleitet werden können.

2.2.2. Benutzerverwaltung

Die Benutzer und deren Rechte werden vollständig im Backend verwaltet. Es muss allerdings bei der Umsetzung der Konfigurationsoberfläche und bei der Umsetzung der Rechte besonders sorgfältig geprüft werden, ob weiterhin der volle Funktionsumfang gewährleistet ist. Sollten bestimmte Rechte für Benutzer oder Gruppen nicht mehr anpassbar sein oder fehlerhaft angewendet werden, ist unter Umständen der gesamte *cRM* nicht mehr vernünftig nutzbar.

2.2.3. Editier-Modus

Änderungen in der Detailansicht werden nicht direkt an die Datenbank übertragen, sondern nur als geändert markiert. Der Benutzer hat dann die Möglichkeit diese zu verwerfen oder den entsprechenden Datensatz zu speichern und die Informationen in die Datenbank zu schreiben.

2.2.4. Individualisierung der angezeigten Elemente

Die Individualisierung der beiden Hauptansichten für Datenentitäten, die Detail- und Listenansicht, ist zur effektiven Visualisierung bestimmter Sachverhalte ausschlaggebend. Bei der Listenansicht können beispielsweise Datenbankspalten ein- bzw. ausgeblendet und frei angeordnet werden, Spalten je nach Datentyp formatiert und spezielle Werte nach selbst definierten Regeln hervorgehoben werden. Die Detailansicht ist noch mächtiger, hier kann jedes UI-Element frei platziert werden, mit Sichtbarkeits- und Formatierungsbedingungen versehen und mit anderen Feldern verknüpft werden (verknüpft bedeutet: Änderung in Feld A löst Änderung in Feld B aus). Um diese Einstellungen vorzunehmen, ist ein Eingabemaskendesigner in die Software integriert, welcher dem Benutzer per „What You See Is What You Get (WYSIWYG)“-Prinzip[1] das Designen erleichtert. Die Weboberfläche sollte diese Funktionalität möglichst vollständig abbilden.

2.2.5. Berichts- und Webansicht

Wie in Abschnitt 2.1 erwähnt, werden Berichte von einem zweiten Produkt der Firma *combit GmbH*, dem sogenannten *LL*, gehandhabt. Dieses besitzt bereits zum jetzigen Zeitpunkt eine entsprechende Weboberfläche, die zukünftig für die Erstellung und das Anzeigen von Berichten genutzt werden kann. Auch die Webansicht kann weiterhin unterstützt werden, indem darin anzuzeigende Ressourcen in ein iframe-Tag[2] eingebunden werden.

2.2.6. Suche / Filter / Sortierung

Das Suchen und Filtern von Datensätzen kann weiterhin mit bestehenden Techniken gelöst werden, indem das Backend Daten vor dem Senden entsprechend im Vorfeld verarbeitet. Da es allerdings sinnvoll ist, Sortierungen in der Übersichtsliste auf dem Client auszuführen — eine Garantie der API, dass Daten immer in einer entsprechenden Reihenfolge gesendet und auch empfangen werden, schränkt das Design dieser zu sehr ein — ist es zumindest überlegenswert, ob das Suchen und Filtern ebenfalls zusätzlich direkt auf dem Client unterstützt werden soll. Viele *JavaScript*-Projekte bieten hierfür bereits fertige und effiziente Lösungen an, welche bei der Umsetzung genutzt werden können.

[1] Echtzeitdarstellung des Ergebnisses während der Erstellung und Bearbeitung
[2] *HTML*-Element, welches das Einbetten von Dokumenten in andere Dokumente erlaubt

2.2.7. Scripting

Die Desktopanwendung des *cRM* unterstützt momentan das Ausführen von Skripten in den Sprachen *VBScript* und *C#*. Für die programmatische Ansteuerung existiert eine Component Object Model (COM)-API, welche im Kontext des Clients, auf dem der Prozess ausgeführt wird, benutzt werden kann. Diese Technologien sind in einer Browserumgebung nicht verfügbar, daher kann ein Skript zwar über die neue UI ausgelöst, aber nicht lokal, sondern ausschließlich im Kontext des Servers ausgeführt werden. Existierende Skripte, welche darauf ausgelegt sind im Kontext des Clients ausgeführt zu werden, müssen vor der Nutzung in der Web-UI entsprechend angepasst werden.

2.2.8. Import / Export

Der Import und Export von Daten in und aus dem *cRM* ist ein wichtiges Feature, wenn es darum geht, verschiedene Programme in einem Workflow zu vereinen. Der *cRM* unterstützt diese Funktion mit einer Vielzahl an Formaten (Excel, Outlook, Datenbanken (ODBC) und Weitere). Diese Funktion kann, wie viele andere auch, weiterhin bestehen bleiben. Dafür muss jedoch eine Möglichkeit geschaffen werden, die Daten über das Netzwerk zur Verfügung zu stellen (Down- und Upload).

2.2.9. Terminverwaltung und Aufgabenplanung

Die interne Termin- und Aufgabenverwaltung ist analog zu anderen Daten über Datenbanktabellen realisiert. Diese können ebenso vom Backend an die Web-UI übertragen und dort dargestellt werden. Der Desktopclient unterstützt aber zudem auch die Verwaltung von Terminen von externen Programmen (z.B. *Microsoft Outlook*). Eine Anbindung der neuen UI an diese kann nur gewährleistet werden, wenn entsprechende HTTP-APIs von den Produkten angeboten werden.

2.2.10. Design für mobile und stationäre Endgeräte

Da die neue Oberfläche alle bisherigen UI-Versionen ablösen soll, ist es ein zentrales Anliegen, sämtliche Endgeräte ihrer Möglichkeiten nach zu unterstützen. Eine bewährte Herangehensweise besteht darin, das mobile Design vor der Desktop-Ansicht zu gestalten. So können die wichtigsten Eigenschaften und Fähigkeiten des Designs direkt angelegt werden, während das Design für größere Bildschirme darauf aufbauend mit zusätzlichen Features, die speziell hierfür sinnvoll sind, ergänzt werden.

Dieses Prinzip wird häufig „mobile-first“-Design genannt. Bei einer umgekehrten Vorgehensweise wird zuerst die Desktop-Ansicht mit allen verfügbaren Features und unter Ausnutzung des gesamten Bildschirmplatzes gestaltet. Dieses Design muss dann für mobile Geräte kompatibel gemacht werden, indem Features und Designelemente entfernt werden. Die Entscheidung, welche Aspekte dabei entfernt werden können, ist schwierig, da sie alle essenziell erscheinen. Es ist zusätzlich auch wichtig zu beachten, dass viele mobile Nutzer — je nach Standort — keine ausreichend gute oder zumindest nur eine teilweise gute Internetverbindung besitzen und es daher wichtig ist, möglichst wenig Daten übertragen zu müssen, bevor eine Webseite angezeigt wird. Das für mobile Geräte charakteristische *CSS* sollte zuerst geladen werde, da dies weitere Downloads von irrelevanten Daten (*CSS* für nicht-mobile Geräte) vermeidet. Für stationäre Geräte ist eine derartige Optimierung jedoch nicht vonnöten.

2.2.11. Vorerst nicht unterstützte Features

Einige aktuelle Features des *cRMs* sind im Web entweder gar nicht oder nur mit sehr viel Aufwand umsetzbar. Diese werden daher vorerst nicht weiter beachtet und deren Umsetzbarkeit eventuell zu einem späteren Zeitpunkt erneut betrachtet:

Scripting-Unterstützung auf dem Client

Skripte werden momentan immer auf dem Client ausgeführt und evaluiert, dabei existiert u.a. Zugriff auf das Dateisystem und andere native Anbindungen des Betriebssystems. Da eine Webseite in der Browserumgebung aus Sicherheitsgründen vom restlichen Betriebssystem isoliert ist, können diese Möglichkeiten nicht genutzt und eine direkte Skriptausführung daher nicht unterstützt werden.

Interaktion mit anderen Prozessen

Bisher war es möglich, mit anderen Prozessen auf dem Client zu interagieren (Starten von externen Programmen, Interkation mit diesen per COM etc.). Ein Beispiel hierfür ist ein Hilfsprogramm, mit dem Anrufe direkt am PC entgegengenommen und automatisch im *cRM* protokolliert werden können. Eine derartige Interaktion ist in einer Browserumgebung, vor allem in einem, von fast allen gängigen Browsern benutzten Sandbox-Modus, nicht möglich.

Ereignisse

Ereignisse sind Events, die zu bestimmten Zeitpunkten der Programmlaufzeit ausgelöst werden und als Aktion zum Beispiel ein Skript starten oder eine E-Mail versenden können. Bevor diese in der Web-UI implementiert werden können, muss zuerst evaluiert werden, welche dieser Ereignisse noch unterstützt werden können (Beispiel: Das Event

„Nach Programmstart“ — anstatt eines Prozesses existiert nur noch eine Webseite. Das Laden dieser ist nicht gleichbedeutend mit dem Starten eines Prozesses, daher kann dieses Ereignis nicht mehr unterstützt werden.)

2.3. Funktionale Anforderungen im Webbereich

Zusätzlich zu den genannten Anforderungen gibt es im Web noch diejenigen Bereiche, die unterstützt werden sollten, um den Nutzern eine besonders gute Erfahrung bei der Bedienung der Seite zu ermöglichen, und solche Bereiche, die den Entwicklungsprozess positiv beeinflussen. Die wichtigstem dieser Anforderungen werden im Folgenden kurz erläutert.

2.3.1. i18n / l10n

Die Akronyme „i18n“ und „l10n“ stehen für Internationalisierung respektive Lokalisierung [vgl. Ishida, W3C, Miller und Boeing 2018]. Gemeint ist damit, dass für jede Sprachressource auf einer Webseite anstelle des Strings ein Identifikator genutzt wird, der mit Strings verschiedener Sprachen verknüpft ist. Je nach Einstellung (Auswahl durch Benutzer, Erkennung der Browsersprache etc.) können dann automatisch alle Texte in der jeweiligen Sprache angezeigt werden. Der *cRM* unterstützt bereits mehrere Sprachen, die Ressourcen hierfür sind allerdings direkt in das Programm eingebettet. Um die gleiche Technologie für die Webseite nutzen zu können, müssten also sämtliche Texte durchweg aus dem Backend angefordert und über das Internet übertragen werden. Dies wäre nicht nur langsam und eine unnötige Belastung für den Server (Texte müssten bei jedem Aktualisieren der UI-Schicht neu geladen werden), sondern auch sehr fehleranfällig. Bei einer unterbrochenen Verbindung könnten überdies keine Fehlertexte angezeigt werden. Besonders im Hinblick auf den weiter unten beschriebenen „offline-first“-Ansatz sollten daher sämtliche Texte in der UI-Schicht gespeichert und jederzeit abrufbar sein.

2.3.2. Barrierefreiheit

Barrierefreiheit von Software bedeutet, dass diese auch von Menschen mit körperlichen Einschränkungen gut genutzt werden kann. Dies ist selbstverständlich immer ein wünschenswertes Ziel, zahlreiche Standards im Webbereich erleichtern ein solches

Vorhaben jedoch in besonderem Maße. So gibt es beispielsweise bestimmte *HTML*-Tags, die extra dafür geschaffen wurden, um von Sprachsoftware vorgelesen zu werden. Weiterhin besteht die Möglichkeit per austauschbarem *CSS* einen „Dark Mode“ anzubieten, sofern ein System von Beginn an entsprechend aufgebaut wurde. Es sollte daher herausgearbeitet werden, welche Möglichkeiten für eine hohe Barrierefreiheit existieren und wie diese bei der Softwareentwicklung beachtet und umgesetzt werden können.

2.3.3. Anleitungs-Modus / „Feature-Tour“

Um neuen Nutzern den Einstieg beim Erlernen eines Produkts zu erleichtern, kann ein interaktiver Anleitungs-Modus, welcher entweder beim ersten Aufruf einer Webseite und / oder bei der ersten Nutzung einzelner Features ausgelöst wird, hilfreich sein. Ein solcher Modus hebt wichtige Bedienelemente hervor und gibt kurze Erklärungen zu ebendiesen, welche entweder direkt durch einen Klick beendet werden können oder intelligent seltener erscheinen, je länger der jeweilige Nutzer das Produkt bereits kennt. Eine mögliche, fertige Umsetzung eines solchen Modus ist in Abbildung 2.4 von *Intro.js* dargestellt.

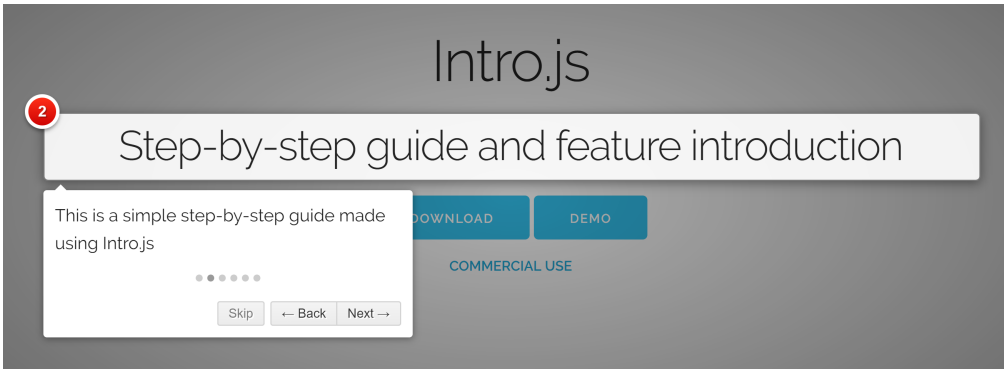


Abbildung 2.4: Anleitungs-Modus von *Intro.js* [Mehrabani o.D.]

2.3.4. „Offline-First“-Ansatz

In jüngster Zeit ist es möglich, mithilfe von Service-Workern[1] und lokalen Speichermöglichkeiten im Browser Webseiten zu erstellen, die auch beim Aussetzen der Internetverbindung zumindest im Wesentlichen weiterhin funktionieren. So können etwa Änderungen, welche an den Server geschickt werden müssen, zwischengespeichert und,

[1] *JavaScript*-Proxy zwischen Webseite und Server; wird in einem anderen Thread als die eigentliche Webseite ausgeführt und kann Anfragen abfangen / verändern

sobald wieder eine Verbindung zum Internet besteht, losgeschickt werden. Ein weiteres Beispiel ist das Anzeigen von Daten, die vom letzten Besuch einer Seite stammen, wenn beim aktuellen Aufruf die Verbindung nicht stabil genug ist. Natürlich sollte ein Nutzer über Änderungen des Status der Internetverbindung informiert werden. Die Information über Verbindungsprobleme sowie die Antizipation ebendieser durch lokales Speichern von relevanten Informationen, zielen beide auf eine für mobile Nutzer möglichst komfortable Bedienung einer Webseite ab, was wie zuvor erwähnt, ein Hauptaugenmerk der neuen Oberfläche darstellt.

2.3.5. Intelligente Fehlerbehandlung

Softwarefehler auf eine für den Benutzer angenehme Art und Weise zu behandeln und im Nachhinein detailliert zu analysieren, ist bei traditioneller Software ein nicht zu unterschätzendes Problem. Bei Webapplikationen kommt jedoch erschwerend hinzu, dass ein Teil der Logik in einer Umgebung ausgeführt wird, welche durch herkömmliche Mittel nicht untersucht werden kann (Browser auf Clientrechner). Umso wichtiger ist es, ein Konzept zu entwickeln, welches in der entsprechenden Umgebung auftretende Fehler sowie Informationen zu deren Lösung übermittelt. Eine Option zur Lösung des Problems bestünde darin, auftretende Fehler immer direkt zum Server zu schicken, um sie vor Ort analysieren zu können. Wenn die eigene Infrastruktur dafür nicht ausreichend ausgebaut ist, gibt es auch entsprechende Online-Dienste, welche die Speicherung eines solchen Fehlerlogs ermöglichen. Zusätzlich dazu müssen auch Parameter für Serveranfragen mit höchster Vorsicht ausgewählt werden: Timeouts müssen lange genug sein, um eine Serverantwort nicht fälschlicherweise zu verwerfen und kurz genug, um bei einem tatsächlichen Fehler die Applikation nicht zu träge wirken zu lassen. Zudem sollten im Fehlerfall erneute Versuche intelligent gestartet werden, das heißt weder zu häufig noch zu selten, um den Server nicht unnötig zu belasten respektive um dennoch einen responsiven Eindruck zu vermitteln.

2.3.6. Nutzung etablierter Authentifizierungsmethoden

Der *cRM* besitzt zwar bereits ein integriertes Benutzersystem mit Authentifizierung und Rechteverwaltung, für die Portierung der Oberfläche sollte aber in Erwägung gezogen werden, diese Rechteverwaltung mit einem oft genutzten, öffentlichen Protokoll wie *OAuth* oder ähnlichen Protokollen zu koppeln, um die Verbindung über HTTP und den Schutz der Authentisierungsdaten nicht mit einer selbst entwickelten, schlechter funktionierenden Lösung handhaben zu müssen.

2.3.7. Caching-Strategien

Um Ressourcen zu sparen, ist es von Vorteil komplexe Berechnungen und zeitintensive Übertragungen nicht mehrmals, etwa bei mehrfachem Laden einer Webseite, auszuführen. Im Bereich des Webs gibt es mehrere Möglichkeiten solche Daten temporär zu speichern: Die einfachste Form besteht darin, Antworten auf eingehende Anfragen auf dem Server und ebenso danach auf dem Client wiederzuverwenden. Wenn der Client selbst kein Caching unterstützt, kann mit „bedingten Anfragen“ dennoch der Cache des Servers genutzt werden: Es wird hierbei eine Anfrage an den Server geschickt und, sollte sich die Antwort im Vergleich zur letzten Anfrage nicht geändert haben, keine Daten, sondern nur ein entsprechender Hinweis an den Client zurückgesendet. Auch das HTTP-Protokoll unterstützt Mechanismen um Caching, beispielsweise über Cache-Proxies, zu erleichtern (spezielle Header, welche die Gültigkeitsdauer einer gespeicherten Ressource bestimmen). Bei allen Formen des Caching ist es jedoch kritisch, dass die richtige Strategie zum Invalidieren der Daten ausgewählt wird, um nicht mit veralteten Werten zu arbeiten, aber dennoch eine Effizienzsteigerung zu erreichen.

2.3.8. Container-Technologien

Container-Technologien wie *Docker* und *Kubernetes* helfen dabei Abhängigkeiten der eigenen Software zu verwalten und auf jedem Entwicklungsrechner eine identische Umgebung zu schaffen. Auch für die Bereitstellung eines Service für Kunden ist es sehr viel einfacher einen bereits korrekt konfigurierten Container zu starten, anstatt eine vollständige Umgebung auf einem Server einzurichten. Dies gilt insbesondere wenn dieser Server von externen Dienstleistern verwaltet wird. Da diese Konzepte bei der *combit GmbH* noch nicht eingesetzt werden, ist es sinnvoll zu untersuchen, an welchen Stellen diese zu mehr Produktivität in Entwicklung und Verteilung der Software beitragen könnten.

2.3.9. Serverless-Konzept

Unter „Serverless“ wird das Konzept verstanden, dass Firmen keine eigenen Server für ihre Kunden oder eigene Zwecke einrichten müssen, sondern auf Lösungen von Drittherstellern (etwa *AWS* von *Amazon* oder *Microsoft Azure*) zurückgreifen können, auf welchen die eigene Software ausgeführt wird. Firmenintern wird also kein Server betrieben, die eigene Infrastruktur ist demnach „Serverless“. Der Vorteil dieses Ansatzes ist, dass die Verantwortung über die Serverhardware nicht selbst getragen werden muss

und die Server durch die großen Kapazitäten der Hersteller je nach Belastung skaliert werden können. Im Gegenzug werden aber eigene Daten fremden Servern anvertraut, je nach Art und Sensibilität der Daten (Betriebsgeheimnisse, DSGVO) kann dies nicht gewünscht sein. Ein solcher Ansatz muss daher gut abgewogen werden, bevor er weiter verfolgt werden kann.

2.3.10. Aktualisierungsstrategie

Kontinuierliche Softwareupdates zur Fehlerverbesserung oder zur Ergänzung neuer Features zählen heutzutage zum Standardservice. Wenn dieser Service angeboten werden soll bedarf es einer kompetenten Strategie, um Modifikationen und Erweiterungen an die Nutzer auszuliefern. Das Backend, auf dem auch die UI-Schicht gehostet werden soll, wird von den Kunden betrieben. Es unterliegt damit nicht dem Verwaltungsbereich der *combit GmbH*, weshalb Updates nicht direkt dort eingespielt werden können. Anstatt Updates für die Oberfläche an Updates des Backends zu koppeln, sollte überlegt werden, ob eine konfigurierbare Option angeboten werden kann, durch die Updateprüfungen auf einem zentralen *combit*-Server ausgeführt werden. Dies setzt allerdings voraus, dass verschiedene Versionen der Oberfläche und des Backend miteinander kompatibel sind.

2.3.11. Integration von bestehenden Ressourcen

Auf der Webseite der *combit GmbH* gibt es bereits zahlreiche Ressourcen wie FAQ oder eine Knowledgebase zu verschiedenen Produkten. Um Nutzern diese Ressourcen präsent zu machen, könnten diese direkt in die Oberfläche, etwa im Rahmen eines Hilfe-Modus, integriert werden.

2.4. Anforderungen an den Entwicklungsprozess

Neben den funktionalen Anforderungen ist es ebenso wichtig, die richtigen Anforderungen für die Projektumgebung und -entwicklung zu formulieren, um diesen Prozess möglichst effizient und skalierbar zu halten. Daher wird empfohlen, die folgenden Technologien und Praktiken bei der Entwicklung des neuen Projektes zu nutzen.

2.4.1. Erstellung von Tests

Alle in Kapitel 3 vorgestellten UI-Frameworks enthalten bereits Lösungen zur Ausführung von Tests. Eine hohe Testabdeckung ist daher nicht nur nützlich, um die Codequalität zu erhalten und Regressionen zu erkennen, sondern auch ohne hohe Kosten und Aufwand erreichbar.

2.4.2. Kontinuierliche Integration

Ein für diesen Zweck geeigneter *TeamCity*-Server von der Firma *JetBrains* ist firmenintern bereits vorhanden, daher sollten Tests von neuen Features immer auch auf diesem Server ausgeführt werden.

2.4.3. Codereviews

Pro Feature sollte auf einem gesonderten Branch gearbeitet und beim Zurückführen in den Masterbranch ein Review durch mindestens einen anderen Entwickler stattfinden, um die Codequalität stets auf einem hohen Niveau zu halten. Auch für Reviews gibt es firmenintern bereits einen *Upsource*-Server (ebenfalls von der Firma *JetBrains*), welcher mit dem *TeamCity*-Server zusammen genutzt werden kann.

2.5. Hauptaugenmerk dieser Arbeit

Eine komplette Umsetzung aller Anforderungen wäre im Rahmen dieser Thesis nicht zu bewältigen. Es gäbe zu viele Aspekte (geeignetes Hosting, eine sichere Authentifizierungsmethode, ein neuer Aktualisierungsmechanismus etc.), die beachtet werden müssen. Summiert reicht die Zeit nicht aus, um Konzepte für all diese Aspekte in einer entsprechenden Qualität und Sorgfalt zu erstellen und umzusetzen. Wie in diesem Kapitel bereits beschrieben, kann die Oberfläche des *cRM* in zwei Hauptbereiche unterteilt werden. Zum einen existieren diejenigen UI-Bereiche, welche sich nicht mit den anzuzeigenden Daten ändern, wie in Abbildung 2.1 links und oberhalb der Detailansicht erkennbar wird (weitere Dialoge und Oberflächen werden kontextbasiert angezeigt), und zum anderen die Detail- und Listenansicht selbst. Die nicht datenabhängigen Elemente und Dialoge bestehen fast vollständig aus statischen Elementen mit teilweise speziellen Anforderungen, welche bei der Umsetzung zwar zeitaufwändig wären, technisch jedoch

kein interessantes Problem darstellten. Es werden im Weiteren daher nur die Oberfläche der Datenansichten (Detail- und Listenansicht) und die damit verbundenen Anforderungen betrachtet. Aus der Liste der funktionalen Anforderungen des *cRM* werden also insbesondere die Automatisierte Erstellung (2.2.1), der Editier-Modus (2.2.3), die Individualisierung (2.2.4) und die Unterstützung der Suche, Filter und Sortierungen (2.2.4) beachtet.

Ein Vorteil von moderner, komponentenbasierter Webtechnologie (alle in Kapitel 3 betrachteten Frameworks sind darauf ausgelegt) ist es, trotz der getrennten Entwicklung verschiedener UI-Bestandteile diese problemlos miteinander zu kombinieren zu können. Es ist sogar möglich unterschiedliche Technologien für die jeweiligen Teile zu nutzen, da zur Laufzeit alle Framework-Umsetzungen in herkömmliches *JavaScript*, *HTML* und *CSS* transpiliert[1] werden und somit miteinander kompatibel sind. Bei dieser zweistufigen Umsetzung kann das Ergebnis dieser Arbeit außerdem bereits in der Webansicht der bestehenden Desktopapplikation angezeigt und als Alternative zu den nativen Oberflächen angeboten werden. So können Kunden dieses Feature bei Interesse bereits im Vorfeld testen, evaluieren und wichtiges Feedback für die weitere Entwicklung einbringen.

[1] Übersetzung von Quellcode aus einer Programmiersprache in eine andere

3

Technologien

Eine den Anforderungen entsprechende Technologie zur Lösung einer Problemstellung auszuwählen ist eine notwendige, jedoch nicht hinreichende Bedingung für Erfolg. Aufgrund dessen sind die auf dem Markt verfügbaren Produkte sorgfältig gegeneinander abzuwägen. Dieses Kapitel beschreibt, welche Technologien für welchen Teil der Architektur in Frage kommen, stellt diese gegenüber und begründet die Entscheidung für die am besten geeignetste Alternative.

3.1. Entwicklungsumgebung

Die Entwicklung des *cRM* findet zurzeit ausschließlich mit *Microsoft Visual Studio* statt. Wie in Kapitel 2 ausgeführt, gibt es die Core- und die Desktopkomponente, welche beide in *C++* geschrieben sind, sowie eine auf den Core aufsetzende, in *C#* geschriebene, *.NET*-Komponente zur Bereitstellung der Webfunktionalität. Eine künftige Entwicklung der Serverfunktionalität für die Weboberfläche kann auf der *.NET*-Komponente aufsetzen bzw. diese erweitern. Dieser Entwicklungsprozess wird nicht verändert und auch nicht weiter thematisiert werden, da die neu zu entwickelnde UI-Schicht eine unabhängige Codebasis besitzt und sich auch konzeptionell von den bisherigen Schichten unterscheidet. Bei der Auswahl der neuen Technologien ist es daher nicht unbedingt notwendig, sich an bestehenden Abläufen zu orientieren.

Das in Kapitel 2 beschriebene Tool zum Einlesen und Konvertieren der bisherigen UI-Repräsentation wurde wegen der vorhandenen, internen Infrastruktur ebenfalls *Visual Studio* verwendet. Als Sprache für die Entwicklung wurde *C#* gewählt, nicht zuletzt

weil diese einen besonders unkomplizierten Zugriff auf das Dateisystem und speziell auf XML-Dateien, das Format der momentanen Repräsentation, erlaubt.

Für die neu entstehenden Webtechnologien kann jeder beliebige Texteditor benutzt werden, da keine speziellen Funktionen einer IDE oder eines Editors notwendig sind. Der in den nachfolgenden Kapiteln entwickelte Code wurde aufgrund der guten Autovervollständigung, der Integration mit Versionsverwaltungstools und der einfachen Erweiterbarkeit, mit *Visual Studio Code* von *Microsoft* erstellt. Wichtiger als der Editor ist die Sprache, für welche er benutzt wird. Im Bereich der Webentwicklung hat sich *JavaScript* ohne Konkurrenz durchgesetzt und ist daher auch die einzige Sprache, die von allen Browsern unterstützt wird. Dies spiegelt sich auch in der Auswahl der Frontend-Frameworks wider, welche ausnahmslos entweder direkt oder indirekt in *JavaScript* geschrieben sind.

3.2. Frontend-Frameworks

Frameworks und Bibliotheken, die versuchen, Entwicklern das Erstellen von UIs zu erleichtern, gibt es schon seit geraumer Zeit und in einer sehr großen Anzahl. Etablierte Projekte wie *jQuery-UI* (Mobile), *Bootstrap* und neuere Projekte (*Materialize*, *Semantic-UI*, *Pure* etc.) setzen auf die Gestaltung von fertigen Komponenten, die per *CSS*-Klasse in eine vorhandene Webseite integriert werden können. Durch diesen Fokus auf Styling mit *CSS* sind diese Projekte meist sehr klein und können schnell integriert werden. Sie eignen sich damit für Prototypen, welche rapiden Änderungen unterliegen, jedoch nicht für die vollständige Erstellung kompletter Webseiten. Sie können vielmehr im Zusammenspiel mit den zuvor angesprochenen *JavaScript*-Projekten benutzt werden.

Im Bereich moderner *JavaScript*-Lösungen sind vor allem komponentenbasierte Frameworks, welchen eine virtuellen DOM[1] zugrunde liegt, stark verbreitet. Aufgrund der zeitlichen Limitierung dieser Arbeit können nicht sämtliche Alternativen gegeneinander abgewogen werden. Es werden daher nur die drei größten Mitstreiter *Vue*, *React* und *Angular* [vgl. Greif, Benitte und Rambeau 2018] miteinander verglichen. Weitere Möglichkeiten für spätere Analysen sind *Polymer*, *Ember*, *Knockout*, *Riot*, und Weitere.

In den folgenden Abschnitten werden verschiedene, bei der Entwickler einer Webapplikation wichtige Gesichtspunkte und deren Umsetzung in den entsprechenden Frameworks beschrieben. Dies dient dem Überblick über die für die Entscheidung relevanten Betrachtungen und zur besseren Nachvollziehbarkeit des im Anschluss gezogenen Fazits.

[1] Virtuelle und meist vereinfachte Repräsentation der UI im Speicher, wird Framework-intern mit dem tatsächlichen DOM der Seite synchronisiert

3.2.1. Generelle Aspekte

Die Entwickler von *Vue* legen besonderen Wert auf die Entwicklung einer möglichst kleinen und dennoch performanten Laufzeit-Umgebung. Das Ziel einer kurzen Einarbeitungszeit sowie einer intuitiven Handhabung wird durch eine sehr gute Dokumentation und ein hervorragendes Command-Line-Interface (CLI)-Tool zum Erstellen neuer Projekte erreicht. Abhängigkeiten zwischen Daten und UI ermittelt *Vue* selbstständig, es werden daher immer nur Elemente aktualisiert, wenn sich auch deren anzuzeigende Daten geändert haben. Das Erstellen von nativen Apps ist bei *Vue* nur mit Software von Drittherstellern möglich. Um diesem Nachteil entgegenzuwirken und den Prozess der Weiterentwicklung zu optimieren, arbeiten die Entwickler eng mit den Drittherstellern zusammen. Das Verwenden von *JavaScript*-Templates (JSX) und *TypeScript* ist mit vorheriger Konfiguration ebenfalls möglich. *Vue* hat ein vergleichsweise kleines Ökosystem, kann dafür aber das stärkste Wachstum aller Frameworks verzeichnen [vgl. Npmjs 2018].

React diktiert im Vergleich zu *Vue* noch weniger Vorgaben an die Entwickler, sämtliche typischen Problemstellungen werden durch Bibliotheken gelöst. Der Hintergrund hierfür ist das umfassende Ökosystem sowie eine breite Auswahl an entsprechenden Bibliotheken. Native Apps (*Android*, *iOS*) können mit dem vom gleichen Team entwickelten *React-Native* erstellt werden, ohne dass sie dafür weitere Software benötigen. Durch die Konzentration auf wesentliche Features und Verbesserungen sowie eine stabile API stellt das *React*-Team sicher, dass Upgrades auf neue Versionen gar keine bis wenige Änderungen am bestehenden Code verlangen. Auch *TypeScript*-Integration und damit verbundenes Tooling (Linter, Typ-Prüfung, Autovervollständigung) sind problemlos möglich.

Im Fall von *Angular* werden zur Erfüllung aller typischen Anforderungen die bei der Entwicklung anfallen, bereits Tools und Konzepte mit ausgeliefert. Dadurch ist *Angular* an sich sehr viel mächtiger als *Vue* und *React*. Die Wahrscheinlichkeit dafür, dass abgesehen von *Angular* noch weitere große Bibliotheken benötigt werden, ist gering. Die Nutzung dieser mitgelieferten Möglichkeiten erfordert jedoch auch, dass ausreichend Kenntnisse über sie vorhanden sind. Diese Notwendigkeit, viele Teilaspekte des Frameworks lernen zu müssen, macht den Einstieg in *Angular* daher relativ mühsam und zeitintensiv. Die Dokumentation ist zwar sehr umfangreich, teilweise aber auch unübersichtlich. Das Ökosystem von *Angular* ist noch größer als das von *Vue*, es hat jedoch auch negatives Wachstum zu verzeichnen [vgl. Npmjs 2018].

3.2.2. Aufbau von Apps (mit Beispielkomponente)

Zur Veranschaulichung der Benutzung der Frameworks wurde eine ausgewählte Beispielkomponente mit jedem der drei Frameworks umgesetzt. In Abbildung 3.1 ist die fertige Komponente dargestellt. Bei einem Klick in das Edit-Feld ändert sich die Farbe des Rahmens, um die Fokussierung zu signalisieren. Wird der Eintrag verändert, kann diese Änderung mit Escape verworfen oder mit Enter übernommen werden. Der Text darunter gibt immer den momentan gespeicherten Wert wieder. Die Komponente wurde in Online-Editoren erstellt, welche bereits eine fertige Umgebung mit dem jeweiligen Framework bieten. Auf die Benutzung weiterer Bibliotheken wurde hier explizit verzichtet.

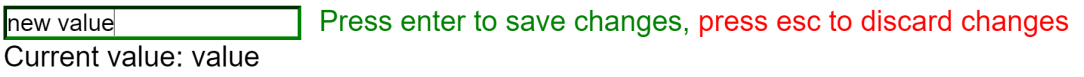


Abbildung 3.1: Beispielkomponente für die Umsetzung mit allen Frameworks

Die *Vue*-Komponente wird vollständig als Instanz einer *Vue*-Klasse angelegt, der Daten und Methoden übergeben werden müssen. Im HTML-Template werden die *Vue*-eigenen Attribute ersichtlich. Eine Implementierung ist in Auflistung 3.1 dargestellt.

```
1  Vue.component('custom-edit-field', {
2      data: () => {
3          return {
4              editing: false,
5              inputValue: '',
6              value: '',
7              borderColor: 'blue',
8          };
9      },
10     methods: {
11         onFocus: function() {
12             this.editing = true;
13             this.borderColor = 'green';
14         },
15         onFocusLost: function() {
16             this.editing = false;
17             this.borderColor = 'blue';
18         },
19         onSubmit: function() {
20             if (this.editing) {
21                 this.value = this.inputValue;
22             }
23         },
24         onAbort: function() {
```

```
25              if (this.editing) {
26                  this.inputValue = this.value;
27              }
28          },
29      },
30      template: `
31          <div>
32              <input
33                  v-on:focusin="onFocus" v-on:focusout="onFocusLost"
34                  v-on:keyup.enter="onSubmit" v-on:keyup.esc="onAbort"
35                  type="text" v-model="inputValue"
36                  class="inputForm"
37                  v-bind:style="{ borderColor: borderColor }"
38              >
39              <span v-if="editing" class="submitInfo">Press enter to save
                    changes, </span>
40              <span v-if="editing" class="abortInfo">press esc to discard
                    changes</span>
41              <div>
42                  Current value: {{value}}
43              </div>
44          </div>
45      `,
46  });
```

Listing 3.1: Umsetzung der Beispielkomponente mit *Vue*

Eine *React*-Komponente kann entweder als herkömmliche Klasse, dargestellt in Auflistung 3.2, oder nur als Funktion, welche die darzustellenden UI-Elemente als Rückgabewert enthält, angegeben werden. Eine Umsetzung der zweiten Umsetzung ist im Anfang B.1 dargestellt.

```
1  class InputValue extends React.Component {
2      constructor(props) {
3          super(props)
4          this.state = {
5              editing: false,
6              inputValue: '',
7              value: '',
8              borderColor: 'blue'
9          }
10         this.input = React.createRef();
11         this.onFocus = this.onFocus.bind(this);
12         this.onFocusLost = this.onFocusLost.bind(this);
13         this.handleKeyPress = this.handleKeyPress.bind(this);
14         this.onChange = this.onChange.bind(this);
15     }
```

```
16      onFocus() {
17          this.setState({editing: true, borderColor: 'green'});
18      }
19      onFocusLost() {
20          this.setState({editing: false, borderColor: 'blue'});
21      }
22      handleKeyPress(e) {
23          if (e.key === 'Enter') {
24              this.setState({value: this.state.inputValue})
25          } else if (e.key === 'Escape') {
26              this.input.current.value = this.state.value;
27              this.setState({inputValue: this.state.value})
28          }
29      }
30      onChange(event) {
31          this.setState({inputValue: event.target.value})
32      }
33      render() {
34          return (
35              <div>
36              <input
37                  onFocus={this.onFocus}
38                  onBlur={this.onFocusLost}
39                  type="text"
40                  className={`inputForm ${this.state.editing ? 'editBorder'
                        : 'normalBorder'}`}
41                  onChange={this.onChange}
42                  onKeyDown={this.handleKeyPress}
43                  ref={this.input}
44              />
45                  {this.state.editing ? <span>
46                          <span className="submitInfo">Press enter to save
                            changes, </span>
47                          <span className="abortInfo">press esc to discard
                            changes</span>
48                      </span> : null}
49                  <div>Current value: {this.state.value}</div>
50              </div>
51          )
52      }
53  }
```

Listing 3.2: Umsetzung der Beispielkomponente mit *React*

Bei *Angular* besteht eine Komponente wie bei *React* aus einer herkömmlichen Klasse. Diese wird jedoch mit einer speziellen Annotation mit Informationen über das Aussehen und die Platzierung der Komponente versehen. Die entsprechende Umsetzung ist in Auflistung 3.3 dargestellt.

```
1  @Component({
2      selector: 'input-element',
3      template: `
4          <input #inputelement class="inputElement" [ngStyle]="{'border-
               color': editing ? 'green' : 'blue' }" (focus)="onFocus($event)
               " (blur)="onBlur($event)" (keydown)="onKeyDown($event)">
5          <span *ngIf="editing" class="submitInfo">Press enter to save
               changes, </span>
6          <span *ngIf="editing" class="abortInfo">press esc to discard
               changes</span>
7          <div>Current value: {{value}}</div>
8      `,
9      styles: [`.inputElement:focus {outline: none;} .submitInfo { color:
           green; } .abortInfo { color: red; }`]
10 })
11
12 export class InputElement  {
13     value = '';
14     editing = false;
15
16     @ViewChild('inputelement') input;
17
18     onFocus() {
19         this.editing = true;
20     }
21
22     onBlur() {
23         this.editing = false;
24     }
25
26     onKeyDown(event: KeyboardEvent) {
27         if (event.key === 'Enter') {
28             this.value = this.input.nativeElement.value;
29         } else if (event.key === 'Escape') {
30             this.input.nativeElement.value = this.value;
31         }
32     }
33 }
```

Listing 3.3: Umsetzung der Beispielkomponente mit *Angular*

3.2.3. Vergleich der Zustands-Verwaltung

Jede Applikation muss früher oder später Daten speichern, also einen gewissen Zustand bewahren, um Nutzern mehr als nur die simpelsten Funktionalitäten anbieten zu können. Wie das Speichern dieser Daten mit den drei Frameworks typischerweise umgesetzt ist, wird in diesem Abschnitt beschrieben.

Der Standard bei *Vue* ist, dass jede Komponente ihr eigenes Zustands-Objekt besitzt. Zusätzlich dazu kann noch auf das Zustands-Objekt der *Vue*-Instanz, in der sich die Komponente befindet, zugegriffen werden. Dieses kann entweder direkt verändert, oder per Store-Pattern[1] verwaltet werden. *Vuex* ist eine direkt von *Vue* entwickelte Bibliothek, welche das genannte Store-Pattern umsetzt.

React unterstützt nur einen unidirektionalen Datenfluss mit sogenannten „Props“[2], welche den Komponenten von ihren Elternkomponenten mitgegeben werden. Jede Komponente hat ihren eigenen Zustand, der per „Props“ an Kinder weitergegeben, aber nicht direkt verändert, werden kann. Es gibt einige bekannte Bibliotheken, welche die Verwaltung von Zustand auf andere Arten lösen. *Redux* setzt beispielsweise das funktionale Flux-Pattern um. Dabei werden Änderungen per Aktions-Event ausgelöst, die Events von sogenannten Reducern ausgewertet und der Zustand von diesen entsprechend verändert. Der neue Zustand wird den entsprechenden UI-Elementen wieder unveränderbar (immutable) übergeben. *MobX* und *react-easy-state* setzen das bei *Vue* erwähnte Store-Pattern in *React* um.

Bei *Angular* wird der Zustand direkt in den Klasseninstanzen der Komponenten gespeichert und kann durch herkömmliche Konzepte der objektorientierten Programmierung weitergegeben und verändert werden. Zusätzlich existiert auch *NGXS*, welches auf demselben Prinzip wie *Redux* beruht, aufgrund der Nutzung moderner *TypeScript*-Features von *Angular* aber ohne weitere Konfiguration einfacher zu benutzen ist.

3.2.4. Vergleich von Routing-Konzepte

Unter Routing versteht man das Konzept, dass bei Navigation zu Subdomänen einer Webseite automatisiert die dafür vorgesehene Komponente dargestellt wird.

Routing ist bei *Vue* direkt integriert. Routen werden in *JavaScript* mit Komponenten verknüpft und bei Aufruf dieser Route wird die verknüpfte Komponente anstatt eines zuvor im DOM platzierten Platzhalters gerendert.

React besitzt keine integrierten Routing-Mechanismen, es gibt aber mehrere Alternativen als Bibliotheken. Bei *Aviator* werden Routen als geschachteltes Objekt

[1] Geteilter Zustand wird zentral verwaltet, Änderungen am Zustand nur intern möglich
[2] Unveränderbare Übergabeparameter

übergeben, bei dem jedes Element eine Route und eine Zielfunktion enthält. Die Zielfunktion wird aufgerufen, wenn zur entsprechenden Route navigiert wird. Sie muss des Weiteren Informationen darüber enthalten, wie und an welche Stelle im DOM (etwa über Dokument-Selektoren) die entsprechende Komponente gerendert werden soll. Die Nutzung von JSX ist dabei nicht mehr möglich. Bei *react-router* hingegen werden die Routen deklarativ in den JSX-Templates hinterlegt. Sie enthalten als Property die Route, für die sie gerendert werden sollen, sowie die eigentlich anzuzeigende Komponente.

Analog zu *Vue* ist bei *Angular* auch eine Lösung direkt integriert welche nach demselben Prinzip funktioniert.

3.2.5. Vergleich der Testintegration

Vue enthält Test-Tools mit denen Komponenten in Variablen gerendert werden können. Diese gespeicherten Komponenten sind herkömmliche *JavaScript*-Objekte und können dann mit weiteren (externen) Tools auf bestimmte Zustände und Eigenschaften geprüft werden. Das Testen von *React*-Komponenten funktioniert analog zu *Vue*.

Sämtliche zum Testen benötigten Tools sind bei *Angular* bereits enthalten und sehr ausführlich dokumentiert. Wenn nur bestimmte Teilaspekte einer Anwendung, wie etwa die UI-Komponenten, getestet werden sollen, ist der Aufwand dies einzurichten aufgrund der Notwendigkeit, sich mit dem gesamten Test-Konzept auseinandersetzen zu müssen, höher als bei *Vue* und *React*.

3.2.6. Vergleich der Server-Side-Rendering-Möglichkeiten

Bei der Benutzung von Server-Side-Rendering (SSR) werden die Seiten nicht auf dem Clientrechner, sondern bereits vor dem Senden auf dem Server gerendert. Es besteht bei allen Frameworks auch die Möglichkeit nur wenige Seiten „vor-gerendert“ auf dem Server abzuspeichern, anstatt sie dynamisch bei entsprechenden Anfragen zu generieren. Hierfür kann das *Node.js* Tool *Prerender.io* benutzt werden.

Vue liefert einen dafür vorgesehenen Renderer (*vue-server-renderer*), der Markup als Text generiert. Dieser Markup-Text kann beim initialen Laden der Seite ausgeliefert werden (erfordert einen *Node.js* Server).

Auch *React* liefert einen dafür vorgesehenen Renderer (*ReactDOMServer*), der Markup als Text generiert. Dieser kann analog zu *Vue* ausgeliefert und auf Clientseite mit „ReactDom.hydrate()“ mit Inhalten gefüllt werden (erfordert einen *Node.js* Server).

Bei *Angular* werden für SSR zusätzliche Abhängigkeiten, Änderungen an der Konfiguration, ein zusätzliches Build / Bundle Target und ein zusätzliches Modul mit separater

Konfiguration, das auf dem Server läuft und dafür zuständig ist, das neue *JavaScript*-Bundle auszuliefern, benötigt.

3.2.7. Fazit

Alle betrachteten Alternativen sind geeignete Optionen zur Umsetzung der zuvor beschriebenen Anforderungen. Bezüglich der Funktionalität bestehen folglich keine Unterschiede, es kommt daher vielmehr darauf an, welches Projekt einem Entwickler (-team) am meisten zusagt bzw. mit welchem er (es) am besten arbeiten kann. Die Einarbeitung in *Angular* kann Schwierigkeiten bereiten, da dort eine Vielzahl an Technologien für verschiedene Zwecke mitgeliefert werden, welche für die Benutzung erlernt werden müssen. *React* besitzt mit Abstand die größte Community, es kann somit davon ausgegangen werden, dass die Weiterentwicklung, zahlreiche aufsetzende Bibliotheken und Ressourcen im Netz garantiert sind. Ein weiterer Vorteil, die Benutzung von *TypeScript* vorausgesetzt, ist die IDE-Unterstützung beim Schreiben der Komponenten-Templates. Im Vergleich zu *Vue* wird bei *React* hier ausschließlich *JavaScript* genutzt, was es erlaubt, optional *TypeScript* einzusetzen und dessen Typisierung zu nutzen, um Autovervollständigung oder ähnliche Hilfen anzubieten. *Vue* hingegen schreibt *JavaScript*-Funktionsaufrufe in Strings und nutzt eigene, für nicht *Vue*-Entwickler unbekannte, *HTML*-Attribute für Schleifen und Verzweigungen. Da es sich dabei um *Vue*-Sonderfälle handelt, kann die IDE oft nicht helfen. Es liegt folglich am Entwickler, das entsprechende Wissen aufzubauen und keine Fehler zu machen.

Nach Abwägung der diskutierten Punkte fiel die Entscheidung für den Prototypen vorerst auf die Nutzung von *React*. Aufgrund der Tatsache, dass die Technologien nach dem Transpilierungsvorgang alle kompatibel sind (sie werden alle in herkömmliches *HTML*, *CSS* und *JavaScript* übersetzt), kann dieser Code später jedoch auch mit *Vue* zusammen genutzt oder allmählich durch alternative Komponenten ersetzt werden.

3.3. TypeScript

Bei *JavaScript* handelt es sich um eine dynamisch typisierte Sprache, das heißt, dass sich Typen von Variablen zur Laufzeit ändern können und vor der Nutzung einer Variablen entsprechend überprüft werden müssen. Dies ist ein Vorteil, wenn innerhalb kurzer Zeit kleinere Skripte geschrieben werden, bei denen aufgrund ihrer überschaubaren Größe entsprechende Überprüfungen trivial sind, oder im Skript enthaltene Fehler vernachlässigbar sind. Für die Entwicklung von größeren Projekten ist diese Eigenschaft

jedoch ein gravierender Nachteil, da Typen von Variablen im Code durch Entfernung von Deklaration und Nutzung nicht ersichtlich sind. Daraus resultierende Fehler treten bei der Benutzung immer erst zur Laufzeit der betroffenen Zeilen auf, oder sind sogar gar nicht als Fehler erkennbar und liefern lediglich ein falsches (aber nicht unbedingt als falsch erkennbares) Ergebnis.

Um dieser Art von subtilen Bugs vorzubeugen, bietet *Microsoft* seit 2012 *TypeScript* an. Es handelt sich dabei um eine typisierte open-source Sprache, welche als syntaktische Obermenge von *JavaScript* (gewöhnlicher *JavaScript*-Code ist also ebenso gültiger *TypeScript*-Code) beschrieben werden kann. Sie wird zu gewöhnlichem *JavaScript* transpiliert und besitzt somit trotz ihrer Vorteile keinen Kompatibilitätsnachteil. Da die Syntax auf *JavaScript* basiert, ist das Erlernen dieser für *JavaScript*-Entwickler trivial. Durch Verwendung typisierter Daten wird die fehlerhafte Nutzungen der Variablen bereits bei der Entwicklung — eine entsprechende Integration des Editors vorausgesetzt — oder spätestens beim Transpilieren bemerkt und dem Entwickler als solche angezeigt. Ebenso erlaubt eine Editorintegration die Bereitstellung einer Codevervollständigung, was Entwicklern häufig die Konsultation der Dokumentation erspart und damit zu einem effektiveren Entwicklungsprozess beiträgt.

Der einzige Nachteil von *TypeScript* ist, dass ein zusätzlicher Schritt zum Übersetzen in *JavaScript*-Code notwendig ist. Die Mehrheit der Projekte im Webbereich setzt jedoch ohnehin vergleichbare Tooling-Schritte voraus, in welche das Kompilieren integrierbar ist, sodass die genannte Einschränkung weniger stark ins Gewicht fällt. Dieser Nachteil kann aber auch positiv ausgelegt werden: durch den Übersetzungsschritt ist es möglich, bereits Features und Standards zu benutzen, welche noch nicht von allen Browsern unterstützt werden. Bei der Übersetzung werden diese in semantisch identischen Code umgewandelt, der unter Einhaltung von älteren Standards gültig ist, aber für den Entwickler schwieriger zu schreiben wäre.

Aufgrund dieser Argumentation soll zur Unterstützung der Entwickler und Vorbeugung von Fehlern bei der Entwicklung der *React*-Seite ausschließlich *TypeScript* benutzt werden.

3.4. API-Anbindung

Für die Anbindung der API kommen zwei Ansätze in Frage: *REST* und *GraphQL*. Bei *REST* handelt es sich um einen Architekturstil, bei dem Ressourcen über URI-Endpunkte angesprochen und abgerufen werden. *GraphQL* hingegen ist eine von *Facebook* entwickelte Query-Sprache zur Abfrage von vorhandenen Daten. In den

folgenden Abschnitten werden speziell die für dieses Projekt relevanten Vor- und Nachteile beider Ansätze angesprochen.

3.4.1. Abwägung von REST für die API-Anbindung

REST-APIs sind heutzutage die Norm, diese Architektur allerdings korrekt umzusetzen, ist jedoch mit viel Arbeit und gewissen Nachteilen verbunden. Dies könnte einer der Gründe dafür sein, dass ein großer Teil der angesprochenen *REST*-APIs nur „REST-like“ sind. Als „REST-like“ werden die Implementationen bezeichnet, welche das sogenannte HATEOAS [1]-Konzept nicht umsetzen. Das Konzept beschreibt, wie ein Client durch die API navigieren soll — alle validen Übergänge vom momentanen Zustand in den nächsten sind bereits in der Antwort einer Anfrage in Form von Links und Metadaten verfügbar. Ein Konsument muss damit nicht mehr wissen, welche Anfragen zu welchem Zeitpunkt gültig sind und kann immer genau die Aktionen anbieten, die auch von der API angeboten werden. Eine Versionierung der API (und damit stärkere Kopplung zwischen Clients und Server) entfällt ebenfalls, neue Features können als weitere Aktion-Links auftauchen und alte, bald nicht mehr verfügbare Features können neben einem Link mit der Information versehen werden, dass sie nicht weiter genutzt werden sollten. Trotz verschiedener Standards, die beim Erstellen der Struktur der Daten helfen, ist es aufgrund des Mehraufwands bei jedem Endpoint dennoch mühsam, eine *REST*-API mit HATEOAS korrekt zu implementieren. Zusätzlich dazu, und dieser Nachteil wird im Gegenzug zu der gerade beschriebenen Flexibilität absichtlich in Kauf genommen, können solche APIs nur navigiert werden, indem vielen Links gefolgt und damit viele Netzwerkanfragen vorgenommen werden. Die Alternative, *REST*-APIs ohne HATEOAS zu erstellen, ist zwar für die Entwickler einfacher, das Konsumieren der API wird dadurch aber deutlich erschwert. Eine Anbindung ist nur durch eine ständige Konsultation der Dokumentation (welche in entsprechender Qualität vorhanden sein muss) möglich. Bei Änderungen an der API müssen auch die Clients angepasst werden, da es keine Möglichkeit gibt sie über diese Änderungen dynamisch zu informieren. Zur Vereinfachung der Erstellung konzeptionell korrekter *REST*-APIs gibt es den *OData*-Standard, der Entwicklern unter anderem die Definition und die Erstellung von Metadaten abnimmt, sodass diese sich auf die eigentliche Programmlogik konzentrieren können.

Caching ist bei *REST* einfacher möglich als bei *GraphQL*: hier kann jedes Endpoint-Daten-Paar als ein Eintrag im Cache angesehen werden. Sind entsprechende Cache-Header in den HTTP-Nachrichten gesetzt, werden die Daten vom Browser (und eventuell vorhandenen Cache-Proxies) automatisch verwaltet. Dies funktioniert, weil Anfragen

[1] Hypermedia as the Engine of Application State

einer Ressource bei *REST* immer alle Daten erhalten, die es zu dieser Ressource gibt. Um nicht immer alle Daten übertragen zu müssen, besitzen viele *REST*-APIs Parameter, mit denen eine Anfrage einschränkt werden kann. Je feiner diese Einschränkungen sein können, desto weniger überflüssige Daten müssen übertragen werden. Gleichzeitig greift damit aber auch der HTTP-Cache immer seltener, weil dauerhaft verschiedene Endpoint-Daten-Paare angefragt werden.

3.4.2. Abwägung von GraphQL für die API-Anbindung

Einer der Hauptgründe für die Entwicklung von *GraphQL* ist die höhere Netzwerklast bei *REST*-APIs, die insbesondere bei mobilen Clients sehr negativ auffallen kann. Ein Hauptaugenmerk der Technologie liegt daher auf der größtmöglichen Sparsamkeit bzgl. der Größe übertragener Daten. *GraphQL* ist laut *npm* die im Webbereich mit am meisten wachsende Technologie überhaupt („hyper-growth“ [vgl. 2018]). Als Folge dessen wird es in Zukunft wahrscheinlich in vielen Projekten benutzt werden, der Aufbau von entsprechendem Know-How ist daher essentiell. Eine Abfrage bei *GraphQL* hat den gleichen Aufbau wie ein JSON-Dokument, mit der Besonderheit, dass anstatt Schlüssel-Werte-Paaren nur Schlüssel bzw. die Namen von Ressourcen angegeben werden. Eine Antwort hat immer den gleichen Aufbau wie eine Anfrage, die Schlüssel bzw. Ressourcennamen werden dabei, wie in Listing 3.4 zu sehen, mit den ausgelesenen Daten ergänzt. Durch die Nutzung von Typen für diese Abfragen kann garantiert werden, dass eine von den Tools akzeptierte Anfrage auf jeden Fall ein Ergebnis vom Server liefern wird. Die Abfragen müssen dabei nicht alle Felder einer Ressource enthalten, es können immer genau die Daten abgefragt werden, die in einer spezifischen Situation gerade benötigt werden. Ebenfalls können mehrere unabhängige Anfragen zu einer großen Anfrage zusammengefasst und mit nur einer Netzwerkanfrage verarbeitet werden. Das typisierte Schema erlaubt weiterhin, Konsumenten jederzeit Informationen über sich selbst (Metadaten), etwa den Typ eines Feldes oder weitere mögliche Felder zur Abfrage, zur Verfügung zu stellen. Vorausgesetzt ein Client kennt einen der Einsprungspunkte eines Graphen, kann er mit diesen Metainformationen alle weiteren Informationen programmatisch auslesen.

```
1 Query:
2
3 {
4   me {
5     name
6   }
7 }
8
```

```
Result:

{
  "me": {
    "name": "Luke Skywalker"
  }
}
```

Listing 3.4: Beispielquery aus der *GraphQL* Dokumentation [GraphQL o.D.]

Um Entwickler zu unterstützen, existiert ein visueller Query-Editor *GraphiQL* mit Autovervollständigung sowie aus dem Schema generierter Dokumentation. Mithilfe dieses Editors kann einerseits sehr leicht in der API navigiert und andererseits können direkt code-seitig benutzbare Abfragen generiert werden. Viele moderne Webseiten basieren nicht mehr nur auf der Annahme, dass irgendwann Daten vom Server abgefragt werden, sondern auch darauf, dass serverseitige Events / Änderungen das Laden oder Anzeigen von Daten auslösen können. *GraphQL* spezifiziert für diese Anforderung ein Abonnement-System: ein Client registriert sich beim Server für alle Daten, über deren Änderung er umgehend informiert werden möchte und gibt an, welche Abfrage dafür ausgeführt werden soll. Bei einer Änderung wird diese Abfrage dann ausgeführt und die Daten über eine Websocket-Verbindung sofort an den Client übertragen.

Um all diese Eigenschaften zu ermöglichen, ist es jedoch notwendig, dass sowohl der Client als auch der Server mit demselben *GraphQL*-Schema arbeiten und damit aneinander gekoppelt sind. Ein weiterer Nachteil ist es, dass aufgrund der Nutzung von *GraphQL* (nur HTTP-POST und identischer Endpoint) Caching nicht auf HTTP-Ebene geschehen kann und damit die Verantwortung für gute Cache-Strategien hauptsächlich beim Client liegen.

Zwei bekannte Bibliotheken, welche die *GraphQL*-Spezifikation in *JavaScript* umsetzen und für die Erstellung der Webseite genutzt werden können, sind *Apollo* und *Relay*. Sie erleichtern das Erstellen von Abfragen („Queries“), Caching und Debuggen bei der Nutzung von *GraphQL*. *Relay* wurde ebenfalls von *Facebook* entwickelt, dennoch hat *Apollo* eine sehr viel größere Community. Für die Umsetzung im Backend (.NET) existieren ebenfalls Implementierungen, die am meisten genutzte davon ist *graphql-dotnet*.

3.4.3. Fazit

REST und *GraphQL* können beide zur Erfüllung desselben Zwecks genutzt werden, sie konkurrieren jedoch nicht direkt miteinander. Eine API kann ebenso beide Ansätze entweder ergänzend oder parallel anbieten. Bestehende *REST*-APIs können von *GraphQL* auf einfache Art und Weise umhüllt und zusammengefasst werden. So können

sie auf beide Arten angesprochen werden. *REST* ist allerdings flexibler als *GraphQL*. Um diese Flexibilität vernünftig zu nutzen und damit eine API zu schreiben, welche viele Jahre genutzt und skaliert werden kann, ist jedoch viel Erfahrung und Aufwand unverzichtbar. *GraphQL* benötigt sowohl eine Server- als auch eine Clientkomponente und hat damit mehr Abhängigkeiten als dies bei *REST* der Fall ist, im Gegenzug ist es dadurch aber möglich, effiziente und typsichere Anfragen zu erstellen. Ebenso wird mit dem mitgelieferten *GraphiQL*-Tool ein Abfrage-Editor mit Autovervollständigung und Fehlerbeschreibungen bereitgestellt, der den Nutzern die Erkundung einer API und deren Möglichkeiten erleichtert, sowie für alle Bestandteile direkt eine Dokumentation generiert. *GraphQL* scheint daher als die sinnvollere Wahl für Firmen oder Personen, die noch nicht viel Erfahrung bei der Erstellung von APIs sammeln konnten. Ein weiterer wichtiger Vorteil von *GraphQL* ist es, dass durch die typisierten Abfragen eine automatische Generierung von Mocking-Daten möglich ist. Mithilfe solcher Mocking-Daten kann das Entwicklerteam den API Client im Frontend entwickeln und testen, bevor das Backend mit den Echtdaten zur Verfügung steht.

Wegen der einfacheren Nutzung von *GraphQL* und der besseren Unterstützung von Entwicklern durch Typisierung und mitgelieferten Tools wird im Prototyp diese Technologie verwendet.

4

Ausarbeitung des Architekturkonzepts

Nachdem in den vorherigen Kapiteln die Anforderungen und zu verwendeten Technologien ermittelt wurden, wird in diesem Kapitel das darauf aufbauende Konzept vorgestellt. Zuerst erfolgt die Beschreibung der neuen Umsetzung bereits vorhandener Konzepte und anschließend die Ausarbeitung der neuen Elemente.

4.1. Analyse und Übersetzung der bisherigen UI-Struktur

Momentan werden alle für die Darstellung der verschiedenen Ansichten benötigten Parameter und Metadaten in einzelnen Dateien als Teil des Solution-Verzeichnis gespeichert. „<Ansichtsname>.dli“ enthält die Konfiguration für die Detailansicht und „<Ansichtsname>.vlc“ die Konfiguration der Übersichtsliste. Für den Aufbau dieser Dateien existiert keine offizielle Spezifikation und auch keine Dokumentation, es ist daher eine Herausforderung, alle nötigen Informationen aus ihnen auszulesen und bedarf unter Umständen spätere Anpassungen des Auslesetools oder der künftigen Struktur. In diesem Abschnitt werden nur die tatsächlich benötigten bzw. ausgelesenen Informationen beschrieben.

4.1.1. Aufbau der Detailansicht

Die relevanten Teile der Detailansicht bestehen aus verschachtelten „List“-Elementen. Die erste Ebene bilden die Seiten („Page0 bis PageN“), eine verschachtelbare Gruppierung von Elementen. Jede Seite besteht aus wenigen Metadaten wie dem Namen, einer

Hintergrundfarbe etc., einem „Controls“-Element und den darin enthaltenen „Control“-Elementen. Ein „Control“-Element enthält Angaben über dessen Typ un abhängig von diesem etliche weitere Eigenschaften, die typspezifisch ausgewertet werden müssen. Die grundsätzliche Form der neuen Struktur ist in Auflistung 4.1 beschrieben.

```
{
    "view": String,
    "pages": Array<{
        "name": String,
        "controls": Array<{
            "type": String, // Static, Edit, Button, ...
            "database_field": String,
            ... // Typspezifische Felder
        }>
    }>
}
```

Listing 4.1: Datentypen der neuen Detailansicht-Struktur

4.1.2. Aufbau der Übersichtsliste

Im Vergleich zur Detailansicht ist der Aufbau der Übersichtsliste simpler. Es existieren eine oder mehrere „List“-Elemente, welche wiederum eine beliebige Anzahl an „Column“-Elementen enthalten können. Ein Listenelement und die enthaltenen Spalten gehören zu einer bestimmten Datenbanktabelle. Diese Information muss also zusätzlich zum Spaltennamen gespeichert werden. Die Form der neuen Struktur ist analog zur Beschreibung der Detailansicht in Auflistung 4.2 zu sehen.

```
{
    "view": String,
    "lists": Array<{
        "database_table": String,
        "columns": Array<{
            "column": String,
            "display_text": String
        }>
    }>
}
```

Listing 4.2: Datentypen der neuen Übersichtslisten-Struktur

4.1.3. Aufbau der neuen Struktur

Die neue generierte Struktur entspricht dem initialen Zustand der neuen UI und enthält neben den ausgelesenen Elementen und deren Formatierung auch Informationen für die Darstellung des Layouts. Die hierfür gespeicherten Informationen entsprechen dabei den von den Layout-Bibliotheken (siehe Abschnitt 4.2.2) generierten Serialisierungen. Das bedeutet, dass weder auf dem Client noch auf dem Server Sonderfälle für die „erste“ (das heißt bevor der Nutzer die Anzeige individualisiert) Darstellung notwendig sind — die Datenstruktur ist identisch, ob es sich um den initialen oder einen angepassten Zustand handelt, ist für Client und Server also vollkommen transparent.

4.2. Konzept der Webseite

Zum jetzigen Zeitpunkt muss die Clientapplikation nur eine einzige Ansicht des *cRM* darstellen können. Hierzu wird ein Projekt mit einer Hauptkomponente und zwei weiteren Komponenten für die Übersichtsliste und die Detailansicht erstellt. Die Hauptkomponente zeigt je nach Auswahl entweder die Listen- oder die Detailkomponente an. In diesen Komponenten wird zuerst die Darstellungsstruktur vom Server abgefragt und anschließend werden anhand der Antwort des Servers vorgefertigte UI-Komponenten dynamisch platziert. Bei externen Änderungen der Daten wird dieser Vorgang mithilfe von *GraphQL*-Abfragen oder -Abonnements (siehe Abschnitt 3.4.2) wiederholt und die UI somit aktualisiert.

4.2.1. React-Komponenten

Die jetzige Oberfläche besteht aus einer festen Anzahl von Darstellungselementen, welche je nach Kontext andere Inhalte anzeigen. Zu den Elementen gehören unter anderem statische Texte, Eingabefelder, Check- und Comboboxen, Gruppierungen und Container für weitere Elemente. Der Kontext für den Inhalt ergibt sich zum einen aus dem Datenbankfeld, das von dem jeweiligen Element dargestellt werden soll, und zum anderen aus Einstellungen wie Sichtbarkeitsbedingungen oder Formatierungen. Da bereits im Voraus bekannt ist, welche Art von UI-Elementen benötigt werden, können diese auch schon im Vorfeld erstellt werden. Diese fertigen *React*-Komponenten werden mit dem Produkt ausgeliefert und können zur Laufzeit dynamisch auf der Webseite platziert werden. Die Funktionalität dieser Elemente orientiert sich dabei immer an der Funktionalität der Originalkomponente. Bedingt durch die Zuständigkeit der Komponenten, die

Abfragen für ihre Daten selbst zu verwalten, haben diese neben der tatsächlichen Darstellung hierdurch eine weitere Aufgabe. Um das Prinzip der eindeutigen Verantwortlichkeit[1] zu bewahren, ist es sinnvoll, die Komponenten in eine Komponente, welche ausschließlich für das Abfragen der Daten zuständig ist, und eine weitere Komponente, welche diese Daten übergeben bekommt und nur für die Darstellung zuständig ist, aufzutrennen. So können beide Bestandteile individuell verändert oder ersetzt werden, was die Wartbarkeit des gesamten Projekts erhöht.

4.2.2. Komponenten-Layout

Für die sinnvolle Nutzung der vorgefertigten Komponenten ist es notwendig, diese nicht nur nacheinander auf der Seite zu platzieren, sondern ein bedienbares Layout für die Platzierung anzubieten. Die Umsetzung wird in diesem Abschnitt dargelegt.

Erste Überlegungen

Zu Beginn stellte sich die zentrale Frage nach der Gestaltung der technischen Umsetzung einer vom Benutzer anpassbaren Anordnung der Detailansicht-Komponenten. Eine naive Herangehensweise ist in Abbildung 4.1 dargestellt. Umgesetzt wurde dies mit einem *CSS*-Grid (roter Rahmen), das entweder horizontal oder vertikal in zwei Hälften getrennt werden kann. Jede dieser Hälften stellt abermals ein *CSS*-Grid dar, welches beliebig zwischen Elterncontainern verschoben werden kann. Es handelt es sich bei diesem Ansatz um eine binäre Baumstruktur an deren Endpunkten (Blätter) sich eine Liste von UI-Komponenten (blauer Rahmen) befindet. Durch den sich hiervon unterscheidenden Aufbau der Desktop-UI ist es schwierig das Layout von dieser auf die *CSS*-Grids abzubilden. Lösungen würden sich vermutlich derart komplex und zeitaufwändig gestalten, dass dieser Ansatz gänzlich verworfen und nach einer alternativen Lösung gesucht wurde.

Umsetzung

Als alternative Lösung zur eigenen Umsetzung der Detailansicht, wurde die Bibliothek *react-grid-layout* gewählt. Zur Integration dieser muss ein Layout in Form von Position, Größe und weiteren Informationen für jede darzustellende Komponente übergeben werden. Anschließend werden alle Kind-Komponenten der Bibliothek entsprechend des Layouts dargestellt. Darüber hinaus besitzt diese Bibliothek einen Editier-Modus, mit welchem das Layout vom Nutzer jederzeit durch Manipulation mit der Maus angepasst werden kann. Je nach Displaygröße des Anzeigegeräts (Smartphone, Tablet etc.) kann die Breite des Layouts so verändert werden, dass weniger Komponenten nebeneinander und stattdessen untereinander angezeigt werden. Außerdem sollen Komponenten

[1] SRP: Single Responsibility Principle

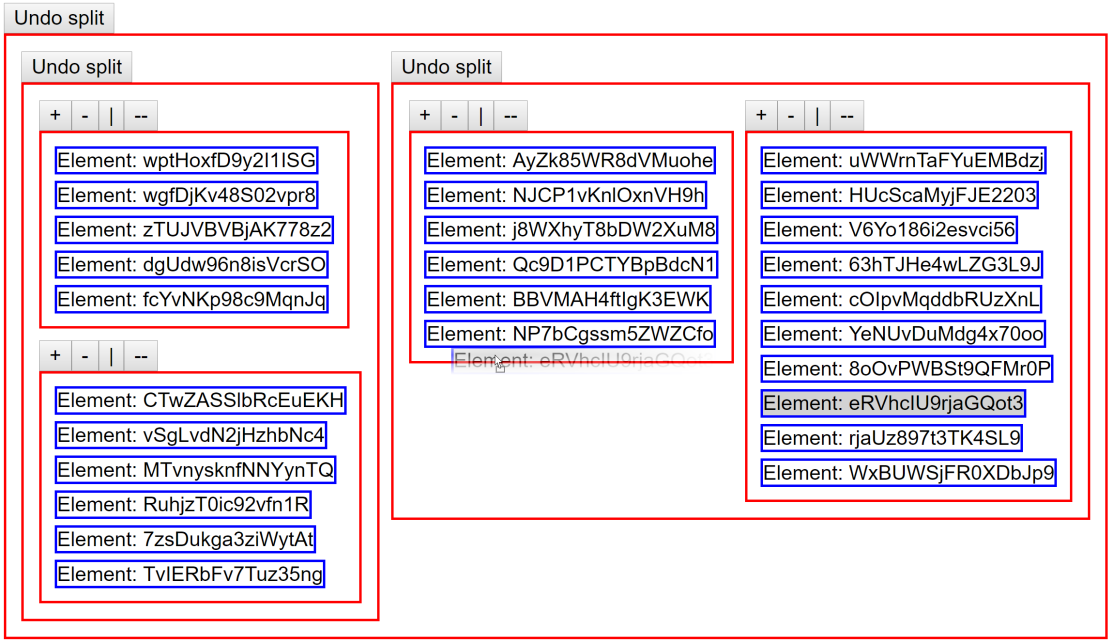


Abbildung 4.1: Eigener Layout-Prototyp mit *CSS*-Grids

eine Konfigurationsoption erhalten, mit der Benutzer die Anzeige für bestimmte Displaygrößen aktivieren und deaktivieren können. Auch für die Übersichtsliste in Tabellenform wird eine passende Bibliothek gesucht. Im Vergleich zur Übersichtsliste werden hier keine eigenen Komponenten angezeigt, sondern die Daten direkt an diese Bibliothek übergeben. Sie muss daher einen gewissen Grad an Flexibilität für die visuelle Anpassung und Formatierung aufweisen. Die am besten geeignete Bibliothek *react-table* befindet sich zum Zeitpunkt der Erstellung dieser Arbeit in umfassendem Umbau, für den Prototyp wird daher die ebenso flexible, aber weniger verbreitete Bibliothek *react-data-grid* genutzt. Sobald der Umbau von *react-table* abgeschlossen ist, kann ein Wechsel hin zu ebendieser diskutiert werden. Für die mobile Darstellung bietet es sich bei einer Tabelle an, sämtliche Spalten einer Reihe zu gruppieren und diese Datensätze in Form von Blöcken untereinander zu platzieren.

4.2.3. Identifikation auf Server

Um die UI-Elemente mit Daten aus der Datenbank darzustellen, muss eine entsprechende Identifikation möglich sein. Es wird vorausgesetzt, dass diese eindeutige ID unabhängig davon, ob sie aus Tabellennamen plus Spaltenname der Datenbank oder aus anderen Informationen besteht, zum Zeitpunkt der Übersetzung einer Ansicht bereits bekannt ist und mit ausgelesen werden kann. Bei Anfragen an den Server werden alle IDs der beteiligten Elemente mit an den Server übertragen, ebenso wie dieser bei

Antworten immer die IDs der Elemente, für welche die Antwortdaten gedacht sind, sendet.

4.2.4. Visualisierung von Lade- und Fehlerzuständen

Direktes Feedback ist für die subjektive Einschätzung einer performanten Webseite essentiell. Typischerweise ist die am längsten dauernde Aktion auf einer Webseite das Nachladen von Daten. Es ist also sinnvoll, diesen Vorgang für Nutzer deutlich und visuell ansprechend zu gestalten. Für diesen Zweck sollen alle *React*-Komponenten eine visuell simplere Repräsentation ihrer selbst in Form von unspezifischen grauen Boxen enthalten, welche nur auf die vage Form und Darstellung mit Echtdaten hindeutet und die bereits während des Ladevorgangs angezeigt werden kann. In Abbildung 4.2 ist eine Gegenüberstellung der beiden Repräsentationen, jeweils für ein Edit-Element und ein Checkbox-Element, zu sehen (finaler Zustand links, Ladezustand rechts).

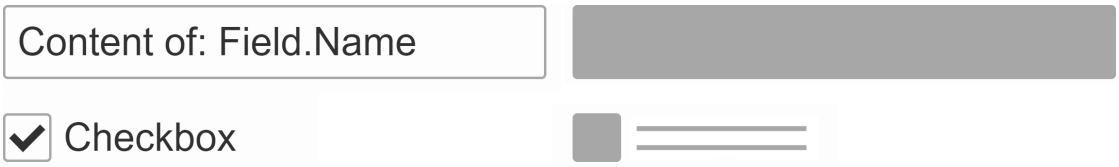


Abbildung 4.2: Ladezustand von Komponenten

Mit einer entsprechende Einfärbung und einem Hinweistext kann diese Visualisierung, wie in Abbildung 4.3 an einer möglichen Ausführung gezeigt, ebenfalls zum Signalisieren von Fehlerzuständen beim Laden der Daten genutzt werden.

Abbildung 4.3: Möglicher Fehlerzustand einer Komponente

4.2.5. Einbindung von GraphQL mit Apollo

Um *Apollo* mit *React* zu nutzen, muss zu Beginn einmalig die Adresse des Servers einer „Apollo-Provider“-Komponente übergeben werden. Diese umschließt alle weiteren Komponenten und dient dazu, die Serveradresse an jeder Stelle der App zur Verfügung zu stellen. Zum Abfragen von Daten für die Darstellung in einer Komponente wird eine spezielle Funktion mit der *GraphQL*-Abfrage, der daraus resultierenden Datenstruktur

und der eigentlichen Komponente konfiguriert. Der Rückgabewert dieser Funktion verhält sich identisch zur übergebenen Komponente, enthält aber zusätzlich noch die aus der Abfrage erhaltenen Daten (und ggf. Lade- / Fehlerinformationen) als Übergabeparameter und kann diese identisch zu lokalen Daten bei der Anzeige nutzen. Dieser Ansatz erlaubt es, die *GraphQL*-Abfragen komponentenspezifisch (jede Komponente ist für den Aufbau der für sie notwendigen Abfrage zuständig) zu schreiben und lokal in einer zugehörigen Datei „<Komponentenname>.gql“ zu speichern. Änderungen oder ein Austausch von Komponenten sind durch diese lose Kopplung der einzelnen Teile der Applikation trivial, da sie nur eine geringe Auswirkung auf alle anderen Komponenten haben.

4.2.6. Nutzung unveränderbarer Daten

Unveränderbare Daten („immutable data“) helfen dabei, Konsistenz und die Nachvollziehbarkeit des Zustands eines Programms zu bewahren. Das Verwalten von Mutationen mit veränderbaren Daten ist schwieriger, je größer ein Projekt wird. Durch falsche Handhabung seitens der Entwickler kann es passieren, dass Änderungen in einem Teil des Programms vollzogen werden und sich automatisch in anderen Teilen auswirken. Wenn diese Auswirkungen unerwartet sind funktionieren Teile des Programms unter Umständen nicht mehr korrekt. Ein weiterer Programmierfehler ist es, bestimmte Änderungen eventuell nicht an jeder notwendigen Stelle zu vollziehen und folglich die Synchronität des Zustand innerhalb eines Programms nicht zu erhalten. Um beide Arten von Fehlern zu verhindern, werden Daten nur an einer zentralen Stelle verwaltet („single source of truth“) und sind an allen anderen Stellen unveränderbar. Mutationen werden als Aktion an die für die Daten verantwortliche Stelle gesendet und nacheinander abgearbeitet.

4.2.7. Konzept des Editier-Modus

In der UI angezeigte Daten kommen direkt vom Server und werden vor der Anzeige nicht weiter verarbeitet (siehe Abschnitt 4.3.2). Aufgrund dieser Tatsache kann eine Kopie des zu editierenden Objekts erstellt und Änderungen auf dieser Kopie ausgeführt werden. Wird der Editier-Modus durch Verwerfen der Änderungen beendet, so muss nur diese Kopie gelöscht werden. Andernfalls wird das Original durch die Kopie ersetzt und der Server über die Mutation benachrichtigt. Dieses Konzept wird von der Bibliothek *immer.js* [Weststrate 2019] umgesetzt und anhand einer Grafik (4.4) sehr anschaulich dargestellt.

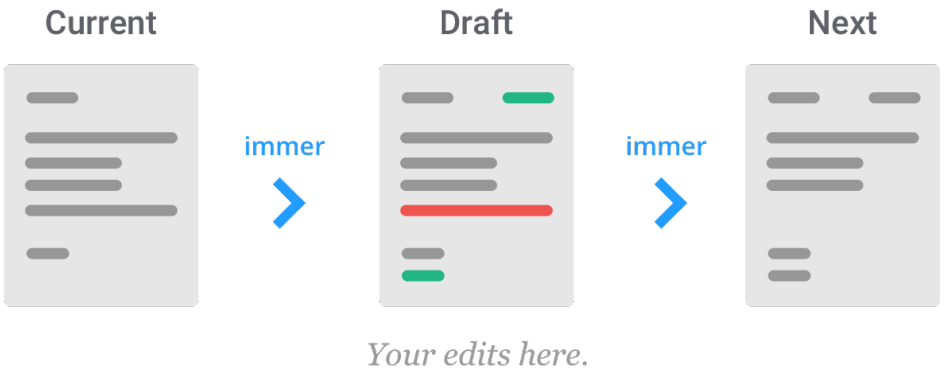


Abbildung 4.4: Editier-Konzept von *immer.js* [Weststrate 2019]

4.2.8. Konzept der Individualisierung

Die Serialisierung des Layouts nach JSON geschieht über die jeweilige Darstellungsbibliothek der Ansicht. Zusätzliche (eigene) Formatierungen aus der Detailansicht werden anschließend im JSON ergänzt und anschließend auf den Server übertragen. Bevor das erste Mal eine solche Anpassung der Ansichten vorgenommen wird, entspricht die Darstellung dabei der in Abschnitt 4.1 beschriebenen, aus der Desktop-UI automatisch generierten Struktur. Das Backend kann diese Informationen entweder direkt als Datei oder in einer NoSQL-Datenbank als Dokument abspeichern, oder seinerseits eine Serialisierung in bestehende SQL-Datenbankschemas vornehmen. Zur Darstellung werden diese Informationen wie in Abschnitt 4.3.1 dargestellt, wieder vom Server abgerufen.

4.2.9. Suche, Filter, Sortierung

Zur Darstellung der Übersichtsliste wird, wie oben beschrieben, die *React*-Bibliothek *react-data-grid* (siehe Abschnitt 4.2.2) verwendet. Diese ermöglicht das Suchen, Filtern und Sortieren ohne weitere Anpassungen. Für die Detailansicht müssen im jetzigen Zustand analog zum Desktopclient die vorhandenen Möglichkeiten im Backend zur Umsetzung der Such- und Filterfunktion genutzt werden. Ein Nachteil dieses Ansatz ist, dass jedes Mal eine Anfrage an den Server geschickt werden und alle (gefilterten) Daten neu übertragen werden müssen. Eine bessere Alternative bestünde darin, die Daten der Detailansicht ebenfalls durch *react-data-grid* verarbeiten zu lassen — die technische Umsetzbarkeit dessen bedarf weiterer Analysen der Bibliothek.

4.2.10. Tests und kontinuierliche Integration

Um Fehler frühzeitig und im Bezug zur der jeweils verantwortlichen Änderung zu erkennen, sollen Tests und kontinuierlichen Integration eingesetzt werden. Boilerplate-Projekte, welche mit dem CLI-Tool *create-react-app (CRA)* erstellt werden, sind bereits so konfiguriert, dass gewöhnliche Tests (mit dem Testframework *Jest*) direkt ausgeführt werden können. Entsprechende Tests sollen dabei entweder schon vor dem Hinzufügen neuer Funktionalitäten oder mindestens parallel dazu stattfinden. Zusätzlich wird noch die Bibliothek *Enzyme* genutzt, welche *React*-Komponenten mit einer beliebigen Verschachtelungstiefe in eine Variable rendern und dadurch in Kombination mit *Jest* sogenannte Snapshot-Tests ausführen kann. Bei dieser Art von Test werden die Komponenten in eine JSON-ähnliche Struktur serialisiert und diese Struktur im Dateisystem gespeichert. Bei jeder weiteren Ausführung werden die alte und die neue Struktur miteinander verglichen und der Test als fehlgeschlagen gewertet, sobald die Strukturen Unterschiede enthalten. Mit dieser Art von Test kann sichergestellt werden, dass Änderungen an der Logik keine visuellen Artefakte generieren. Alle herkömmlich sowie die beschriebenen Snapshot-Tests können im Sinne der kontinuierlichen Integration auf dem internen *Team City*-Server ausgeführt werden. Neben diesen Tests wird während der Entwicklung und Anpassung der UI das Tool *Storybook* genutzt, mithilfe dessen eine *React*-Klasse isoliert von allen anderen Klassen angezeigt werden kann. So ist es möglich, eine interaktive Echtzeit-Darstellung von Änderungen am Code zur direkten Beurteilung der gewünschten Auswirkungen zu nutzen.

4.3. Konzept der API

GraphQL spezifiziert drei grundsätzliche Anfragetypen: Abfragen („Query“), Mutationen („Mutation“) und Abonnements („Subscriptions“). Für die API existieren aufgrund des fehlenden Backends, auf dem Mutationen an Daten vorgenommen werden können, momentan hauptsächlich Abfragen. Mutationen können aber eine annähernd identische Struktur wie Abfragen nutzen, es muss für jede veränderbare Ressource lediglich ein Parameter im Schema aufgenommen werden, der den neuen Wert der Ressource repräsentiert. Abonnements werden in der aktuellen Version noch nicht genutzt, da zuerst eruiert werden muss, an welchen Stellen Echtzeitupdates notwendig sind und an welchen herkömmliche Abfragen in bestimmten Zeitintervallen ausreichen. Wenn Abonnements in Zukunft genutzt werden, dann lediglich um den Client über die Art des Updates in Kenntnis zu setzen und nicht, um direkt sämtliche Daten erneut zu senden. Dies hat

den Vorteil, dass ausschließlich Abfragen zum Laden von Daten genutzt werden und die Clientapplikation selbst entscheiden kann, zu welchem Zeitpunkt die aktualisierten Daten angefragt werden.

4.3.1. Aufbau des GraphQL-Abfrage-Schemas

Das Schema, welches den Aufbau der Anfragen und damit gleichzeitig der Antworten vorgibt, enthält mehrere „Einstiegspunkte“[1]. Eine Anfrage aller Informationen und deren Antwort kann in den Anhängen A.1, A.2 und A.3 eingesehen werden.

Es besteht die Möglichkeit Informationen zur Struktur der Übersichtsliste, der Detailansicht und die eigentlichen Daten abzufragen. Clients können so zuerst die Struktur, das heißt sämtliche enthaltenen UI-Elemente und deren Platzierung, Formatierung etc. einer Ansicht abfragen. Die reine Struktur enthält vergleichsweise wenige zu übertragende Daten, eine entsprechende Abfrage bekommt demnach zügig eine Antwort übermittelt. Während die UI anschließend anhand der Struktur aufgebaut und Lade-Platzhalter (siehe Abschnitt 4.2.4) angezeigt werden, kann parallel das Nachladen der Daten stattfinden. Sobald auch dieser Vorgang abgeschlossen ist, werden die Lade-Platzhalter je nach Erfolg beziehungsweise Misserfolg mit den tatsächlichen Feldinhalten oder der Fehlervisualisierung inklusive Fehlermeldung ausgetauscht. Das Abfrageschema für die beiden Struktureinstiegspunkte ist an die Anforderungen des Clients angelehnt und liefert die Daten in einer Form, die der Hierarchie der *React*-Komponenten entspricht. So können die für die entsprechenden Komponenten relevanten Teildaten ohne weitere Bearbeitung übergeben werden. Der Dateneinstiegspunkt ist derart aufgebaut, dass entweder alle Datensätze und deren Felder oder nur ein spezifischer Datensatz, bestimmt durch dessen eindeutige ID, abgefragt werden können. Die Feldliste kann dabei ebenso per Feldname-Parameter eingeschränkt werden. Um außerdem neue Elemente zur Oberfläche hinzufügen zu können, müssen sämtliche zur Verfügung stehenden Felder aus dem Backend angeboten werden können. Dazu soll die API zusätzlich zu den bereits erwähnten Eigenschaften die Möglichkeit besitzen, eine Liste dieser Felder sowie pro Feld(typ) alle UI-Elemente, welche in der Lage sind die Daten des Feldes anzuzeigen, abzufragen.

[1] Wurzel, an dem eine *GraphQL*-Abfrage ansetzt und von der aus relativ weitere Punkte im Datengraph abgefragt werden können

4.3.2. Verhinderung von Business Logik im Client

Aufgrund verschiedener Interpretationsmöglichkeiten sollten Clients keine eigenen Schlussfolgerungen aus den von der API gelieferten Daten ziehen müssen. Dies ist ein häufig begangener Fehler, welcher darin besteht, dass die API Daten zu einer Ressource angibt, die jedoch nicht direkt angezeigt werden. Stattdessen müssen diese Daten, bevor sie angezeigt werden können, zunächst noch verarbeitet und transformiert werden. Es kann vorkommen, dass verschiedene Implementierungen von Clients an dieser Stelle andere Transformationen anwenden und die sichtbare Darstellung dabei für Endanwender inkonsistent erscheint. Selbst wenn darauf geachtet wird, dass zu einem Zeitpunkt X alle Clients konsistent implementiert sind, ist es möglich diese Konsistenz durch Bugfixes oder andere clientspezifische Anpassungen in der Zukunft zu einem späteren Zeitpunkt Y wieder zu verlieren. Ein Beispiel für einen solchen Fall, beschrieben von *Phil Sturgeon* [vgl. 2017], ist eine API, die Anfragen über Rechnungen beantwortet, und dabei verschiedene Felder liefert. Enthält das Feld **bezahlt-am** keine Daten, können Clients davon ausgehen, dass die Rechnung noch nicht bezahlt wurde. Wird nun ein weiteres Feld, welches anzeigt, ob der Betrag tatsächlich auf dem Empfängerkonto eingegangen ist, **bezahlung-erhalten-am** hinzugefügt, zeigen alte Clients den Status weiterhin als „bezahlt“ an, insofern das erste Feld einen Wert enthält. Clients, die ein Update erhalten haben, zeigen diesen Status jedoch erst, wenn beide Felder einen Wert enthalten. Entsprechend ist diese API so konzipiert, dass notwendige Transformationen immer im Backend vollzogen werden. In der aktuellen Implementierung gibt es noch keine Instanz, bei der dieses Paradigma angewendet werden muss, spätere Anpassungen und Ergänzungen an der API müssen aber entsprechend umgesetzt werden.

5

Implementierung eines Prototypen

In diesem Kapitel wird die Entwicklung eines auf dem zuvor ausgearbeiteten Konzept basierten Prototypen beschrieben. Dieser Prototyp enthält nicht alle im Konzept beschriebenen Anforderungen und befindet sich auch nicht in einem finalen Entwicklungsstadium, kann aber mit weiteren Entwicklungsressourcen als Grundlage für eine finale Implementierung genutzt werden. Er soll zeigen, wie sich das entworfene Konzept umsetzen lässt und die darin benutzten Technologien miteinander interagieren.

5.1. Tool zum Parsen der vorhandenen UI-Struktur

Wie in Kapitel 4 beschrieben, müssen die Dateien, welche das momentane UI-Layout enthalten, in ein web-freundliches Format (JSON) übersetzt werden. Während dieses Prozesses können künftig nicht mehr benötigte Informationen übergangen, das heißt nicht mit in das neue Format übernommen, und relevante Informationen direkt in eine optimierte Struktur überführt werden. Für diesen Zweck wurde ein kleines Hilfstool in *C#* geschrieben, welches sowohl die „.dli“-Datei der Detailansicht als auch die „.vlc“-Datei der Übersichtsliste einer einzelnen *cRM*-Ansicht als Input erhält und daraus eine „.json“-Datei mit allen benötigten Informationen erstellt. Um die Anpassbar- und Wiederverwendbarkeit des Tools zu maximieren, wurde das **Visitor-Pattern** angewandt. In Abbildung 5.1 ist der Aufbau des Tools dargestellt.

Zunächst wird für jedes einzulesende Token (XML-Element) eine Klasse vom *Acceptor*-Interface (*IDLIAcceptor* / *IVLCAcceptor*) abgeleitet, dieses Interface enthält eine einzige Methode *Apply*, welche einen Visitor (*IDLIVisitor* / *IVCLVisitor*) übergeben

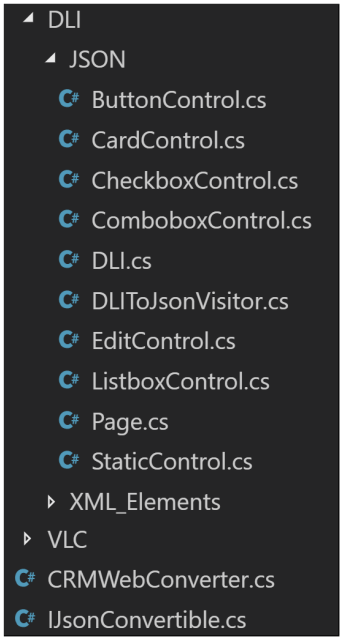


Abbildung 5.1: Datei-Struktur des Konvertierungstools

bekommt. Die jeweiligen Token-Klassen werden mit dem von ihnen verwalteten XML-Element (*XElement*) initialisiert. Wie im Quellcodeauszug 5.1 anhand der Klasse *PageElement* beispielhaft zu sehen, werden aus diesem XML-Element die relevanten Informationen über das Element selbst (Name und Metadaten) und dessen verschachtelte Kinder-Elemente (Liste von Controls) ausgelesen.

```
1  public PageElement(XElement element) : base(element)
2  {
3      Name = element.Attribute("name").Value;
4      foreach (var childElement in element.Elements())
5      {
6          if (CheckAttribute(childElement, "name", "Title"))
7          {
8              MetaData = childElement.Value;
9              break;
10         }
11
12         if (CheckAttribute(childElement, "name", "Controls"))
13         {
14             ControlList = new ControlListElement(childElement);
15         }
16     }
17 }
```

Listing 5.1: Initialisierung der PageElement-Klasse

Nachdem die Informationen der XML-Datei auf diese Art und Weise in ihre einzelnen Token-Instanzen übersetzt wurden, wird die *Apply*-Methode, zu sehen in Quellcodeauszug 5.2, des zentralen Tokens (*DialogElement*) mit einer Visitor-Instanz aufgerufen. Der Visitor erhält als Parameter eben diese Instanz und extrahiert alle für ihn relevanten Informationen. Anschließend ruft er rekursiv die *Apply*-Methoden der Kinder-Token auf und liest auch aus diesen die relevanten Informationen aus. Diese Aufrufe sind im Quellcodeauszug in Anhang C.1 zu sehen. Nachdem alle Tokens vollständig besucht wurden, können die gewonnenen Daten als JSON-String ausgegeben werden.

```
1 public override void Apply(IDLIVisitor visitor)
2 {
3     visitor.Visit(this);
4 }
```

Listing 5.2: *Apply*-Methode der DialogElement-Klasse

Die Flexibilität dieser Architektur, welche in der Abbildung 5.2 nochmals übersichtlich als Klassendiagramm dargestellt wird, ist des Weiteren daran zu erkennen, dass der einzige Unterschied beim Auslesen von Detailansicht-Datei und Übersichtslisten-Datei in der Implementierung der Interfaces besteht. Sowohl *Acceptor*- als auch *Visitor*-Klassen können sehr leicht einzeln angepasst oder ersetzt werden. Ebenso besteht die Möglichkeit, weitere Tokens, welche eventuell zu einem späteren Zeitpunkt benötigt werden, aus den XML-Dateien auszulesen, indem weitere *Acceptor*- und *Visitor*-Implementierungen hinzufügt werden.

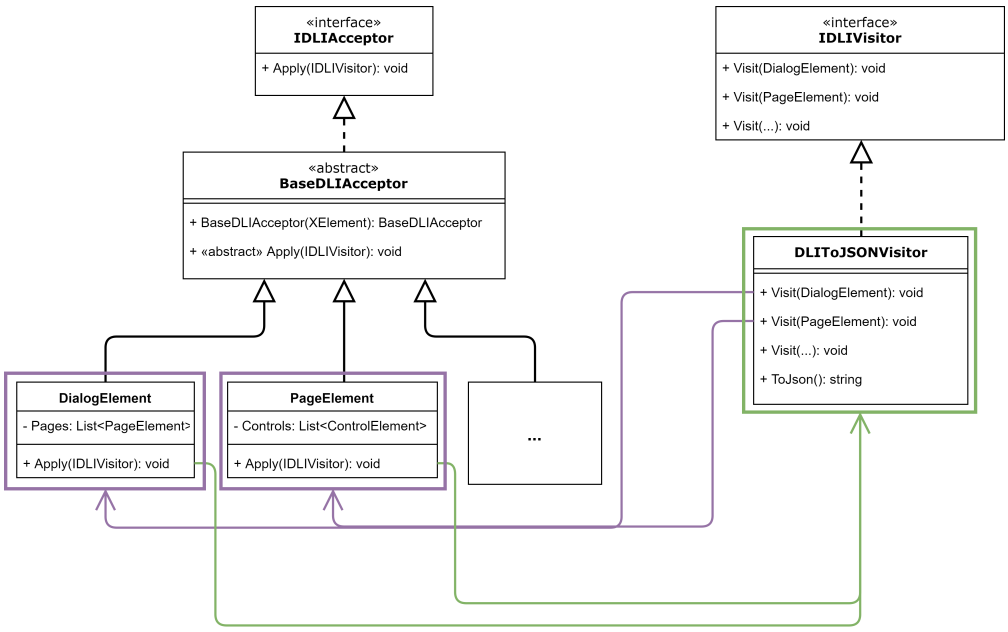


Abbildung 5.2: Klassendiagramm der Visitor-Struktur des Konvertierungstools

Der Input und das Endergebnis in Form eines vom *GraphQL*-Server direkt verwertbaren JSON-Dokument ist im Quellcodeauszug 5.3 und Auszug 5.4 anhand eines kleines Auszuges ersichtlich.

```
<list name="Page0">
    <list name="Controls">
        <list name="Control">
        <item name="TypeIndex">1</item>
        <item name="Type">STATIC</item>
        <item name="Rect(l,t,w,h)">(18,28,420,11)</item>
        <item name="ColorBackground">Transparent</item>
        <item name="CondVisible">Deactivated</item>
        <item name="InputLCID">0</item>
        <item name="EnvironmentRestriction">0</item>
        <item name="UIModifiers">0:UkdCKDI1NSwwLDAp,1:IlNlZ29lIFVJIg==,2
            :RmFsc2U=,3:RmFsc2U=,4:RmFsc2U=,5:RmFsc2U=,6:MTMuMA==</item>
        <item name="Font">{(255,0,0),13,-17,0,0,0,400,0,0,0,0,3,2,1,34,
            Segoe UI}</item>
        <item name="Calculated">TRUE</item>
        <item name="Formula">«&quot;--- deaktiviert seit &quot; + Date$(
            Date(DeactivatedOn),&quot;%02d.%02m.%04y %02H:%02i:%02s&quot;)
             + Cond(not IsNullOrEmpty(DeactivatedReason), &quot;, Grund: &
            quot; + DeactivatedReason) + &quot; ---&quot;»</item>
        <item name="Orientation">0</item>
        </list>
        ...
```

Listing 5.3: XML-Input

```
"pages": [
    {
    "name": "Page0",
    "title": "",
    "controls": [
        {
        "type": "static",
        "text": "",
        "textFromServer": true
        },
        ...
```

Listing 5.4: JSON-Ergebnis

5.2. Umsetzung der Webseite

Das Projekt für die Webseite wurde mit dem von *Facebook* entwickelten CLI-Tool *create-react-app* erstellt. Dieses führt die allgemeine Konfiguration, die Einrichtung von *TypeScript* und der Test-Funktionalitäten selbstständig durch, weitere Anpassungen sind daher nicht notwendig. Neben den in diesem Abschnitt beschriebenen Komponenten und Tests enthält das Projekt ausschließlich die Definition zweier verschiedener *CSS*-Themes und einige wenige temporäre UI-Elemente zur Steuerung des Verhaltens (Umschalten des Anzeigemodus, des Themes und des Editier-Modus).

5.2.1. Erstellung der React-Komponenten

Der Aufbau aller Komponenten folgt einem identischen Schema. Zunächst ist eine *React*-Komponente so definiert, dass sie als Parameter alle notwendigen Eigenschaften zur Anzeige und Mutation von Daten übergeben bekommt. Die Komponente enthält keine Logik zur Veränderung der Parameter. *CSS* wird mithilfe der Bibliothek *styled-components* in Form eines Wrappers um das eigentlich anzuzeigende *HTML*-Tag eingesetzt. Ein Nachteil des traditionellen *CSS*-Einsatzes ist, dass *CSS*-Klassen an vielen Stellen der Oberfläche eingesetzt werden und eine Änderung am Aussehen durch den fehlenden Überblick über sämtliche Stellen möglicherweise ungewollte Auswirkungen haben kann. Durch die Verknüpfung von *CSS* mit *React*-Komponenten kann sich eine Änderung immer nur auf die jeweils verknüpfte Komponente auswirken. In Ausschnitt 5.5 ist die Benutzung und Unterstützung von Themes von *styled-components* anhand eines *HTML* input-Tags dargestellt. Eine Komponente ist nicht selbst für das Abschicken der *GraphQL*-Abfragen, welche die Struktur der Ansicht liefern, verantwortlich. Dennoch wird der jeweilige Teil der Abfrage mit den für sie relevanten Daten in der Projektstruktur zusammen mit der Komponente definiert. Eine speziell dafür ausgelegte Komponente in einem höheren Teil der Hierarchie kombiniert sämtliche Teilabfragen der UI-Komponenten (siehe Ausschnitt 5.6) und sendet diese an den Server. Diese Komponente ist weiterhin dafür zuständig, die vom Server erhaltenen Daten wieder auf die UI-Komponenten zu verteilen. Für die Abfragen der tatsächlichen Daten wird eine weitere Abfrage für jede Komponente definiert, welche jeweils auch von dieser Komponente ausgeführt und ausgewertet wird. Der vierte und letzte Bestandteil einer Komponente sind die im nächsten Abschnitt beschriebenen Tests.

```
// ThemedEdit can be used like a regular React component
const ThemedEdit = styled.input`
    // properties depending on theme (dynamic)
    color: ${(props: SimpleThemeProps) => props.theme.color};
    // static css property
    width: 100%;
    // additional css properties
    ...
`;
```

Listing 5.5: Benutzung der *styled-components* Bibliothek

```
// query fragment of Edit component
const EditFragment = gql`
  fragment EditControl on Controls {
    ... on Edit {
      field
    }
  }
`;

// embedding of fragments
 const ControlsFragment = gql`
  fragment Controls on Page {
    controls {
      ...EditControl
    }
  }
  ${EditFragment}
`;
```

Listing 5.6: Einbettung von *GraphQL*-Query-Fragmenten

5.2.2. Integration der Tests

Einzelne Komponenten werden durch die in Abschnitt 4.2.10 erwähnten Snapshot-Vergleiche (siehe Ausschnitt 5.7) auf unerwünschte Veränderungen geprüft. Weiterhin wird getestet, ob die Übergabeparameter in Form von Formatierungsbedingungen oder anzuzeigenden Daten korrekt ausgewertet und dargestellt werden. Da die Clientapplikation bewusst wenig Logik enthält, werden in diesem Bereich auch verhältnismäßig wenige Tests benötigt. Die während der Entwicklung genutzten visuellen *Storybook*-Tests müssen für jede einzelne Komponente konfiguriert werden. Notwendig ist lediglich die Konfiguration der Komponente selbst. Optional werden aber jeweils noch die im

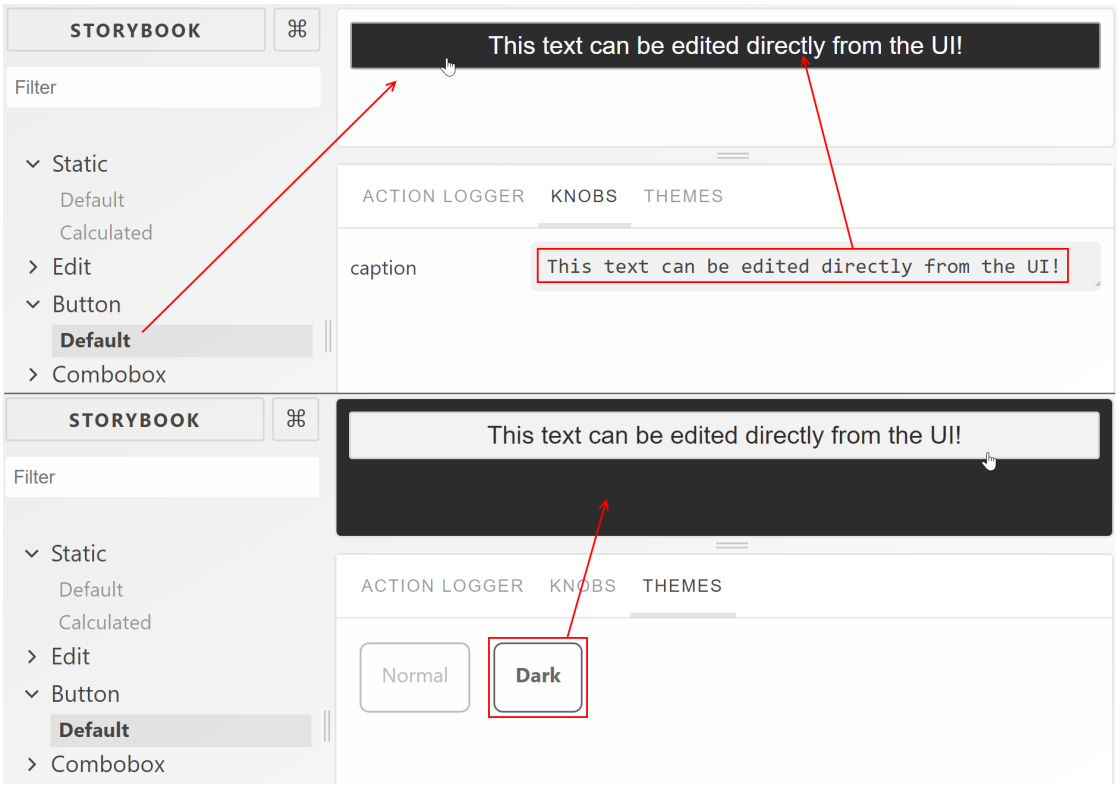


Abbildung 5.3: Beispiel für die Komponentendarstellung mit *Storybook*

Produktivcode ebenso benutzten globalen *CSS*-Deklarationen, die verfügbaren Themes und die in der UI veränderbaren Übergabeparameter (im Ausschnitt „Knobs“) angegeben. Abbildung 5.3 zeigt die Darstellung der in Ausschnitt 5.8 gezeigten Konfiguration in der *Storybook*-Umgebung.

```
1 test('Button should match snapshot', () => {
2     expect(mount(<Button id={buttonId} caption={buttonCaption} />)).
          toMatchSnapshot();
3 });
```

Listing 5.7: Snapshot-Test mit *Jest* und *Enzyme*

```
1 storiesOf('Button', module)
2     .addDecorator(globalStyleDecorator)
3     .addDecorator(withThemesProvider([normalTheme, darkTheme]))
4     .addDecorator(withKnobs)
5     .add('Default', () => <Button id={'1'} caption={text('caption', 'Some
          text')} />);
```

Listing 5.8: Konfiguration einer *Storybook*-Komponente

5.2.3. GraphQL-Mock-Server und Resolver

Da das passende Backend zu dieser Arbeit erst zu einem späteren Zeitpunkt erstellt wird, wurde vorübergehend eine Dummy-Implementierung eines *GraphQL*-Servers erstellt, welcher aber bereits das korrekte Schema und die Daten des erstellten Parser-Tools nutzt. Dies ermöglicht es, den Client unter realen Gegebenheiten zu entwickeln. Die Implementierung besteht aus der Schemadefinition für *GraphQL* (ein Auszug daraus ist in Quellcodeauszug 5.9 zu sehen) und den Resolvern, welche dem Schema die konkreten Daten zuordnen (der zu Auszug 5.9 passende Code kann in Auszug 5.10 eingesehen werden). Aus diesen beiden Einzelteilen wird mithilfe der *Apollo GraphQL-tools* [2019] ein ausführbares Schema erstellt und mit *express-graphql* [GraphQL 2018] lokal gehostet. Für die im Schema definierten Typen, welche aufgrund fehlender Daten noch keine Resolver erhalten können, werden von den *GraphQL-tools* automatisch Daten mit passenden Typen generiert.

```
type Page {
    title: String
    controls: [Controls]
}

union Controls = Static | Edit | Button

type Static {
    text: String
    fromServer: Boolean
}
```

Listing 5.9: Teil der *GraphQL*-Schemadefinition

```
Page: {
    title: (parent: { title: string }) => parent.title,
        controls: (parent: { controls: {}[] }) => parent.controls
    },
    Controls: {
        __resolveType(obj: { type: string }) {
        if (obj.type === "static") {
            return "Static";
        }

        if (obj.type === "edit") {
            return "Edit";
        }

        if (obj.type === "button") {
            return "Button";
```

```
17          }
18
19          return null;
20          }
21      },
22      Static: {
23          text: (parent: { text: string }) => parent.text,
24          fromServer: (parent: { textFromServer: boolean }) => parent.
                textFromServer
25      }
26  }
```

Listing 5.10: *GraphQL* Schema-Resolver

5.3. Beispielumsetzung

Um die Validität des Konzepts zu belegen, wird eine beispielhafte, simple Oberfläche des *cRM* Desktopclients in die neue Darstellung übersetzt. In den Abbildungen 5.4 und 5.5 ist die bisherige sowie die neue Übersichtsliste zu sehen. Die angezeigten Spalten der neuen Liste entsprechen dabei der Konfiguration der Liste auf dem Desktopclient.

Simple

Company	Country	ZIP	City	Street	Phone	Email	ModifiedOn	ModifiedBy
Smith Real Es...	US	10010	New York	132 E 23rd St	212-557/099...	info@relatio...	22.11.2018 ...	Administrator
Weinhandlun...	DE	78462	Konstanz	Schottenstr. ...	07531/099992	info@relatio...	17.11.2018 ...	THeld
Gas- und Wa...	DE	78467	Konstanz	Opelstr. 15	07531/0999...	info@relatio...	17.11.2018 ...	THeld
Herrenbeklei...	DE	66111	Saarbrücken	Landwehrplat...	0681/0999999	info@relatio...	22.11.2018 ...	THeld

Abbildung 5.4: Übersichtsliste des Desktopclients

Show list view Use dark theme

ListView of 'Simple':

Company	Country	ZIP	City	Street	Phone	Email	ModifiedOn	ModifiedBy
voluptatem	eos sint	et	ab quo occaecati	vel incidunt	et perferendis	natus animi	unde	optio fuga est
ratione consequatur	et quos	quo voluptatem corr	mollitia	voluptatum eos	nesciunt dolorem ips	ut repellendus	incidunt rem	et est
est in	voluptatibus exercita	voluptatem	aut magnam	quod	voluptates ullam	voluptas ut aliquam	ullam quis	mollitia eos
fuga est	et error	nihil cum	atque	veniam dolorem	quisquam est	voluptas id quia	ut	fugiat nesciunt adipi

Abbildung 5.5: Übersichtsliste der Weboberfläche

Analog zur Übersichtsliste stellen die Abbildungen 5.6 und 5.7 die Detailansicht des Desktopclients und der Web-UI mit aktiviertem „Dark Mode“ dar. Das Layout der Weboberfläche entspricht unmittelbar nach der automatisierten Umsetzung einer Liste von nacheinander angezeigten Elementen. Sie wurde für die Darstellung daher bereits manuell angepasst, um der des Desktops zu entsprechen.

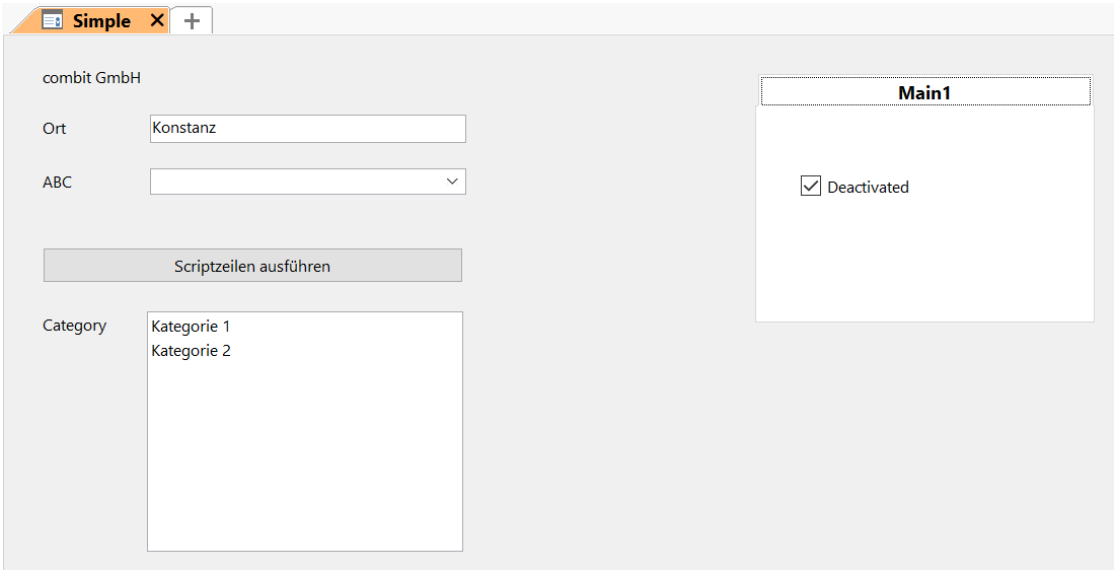


Abbildung 5.6: Detailansicht des Desktopclients

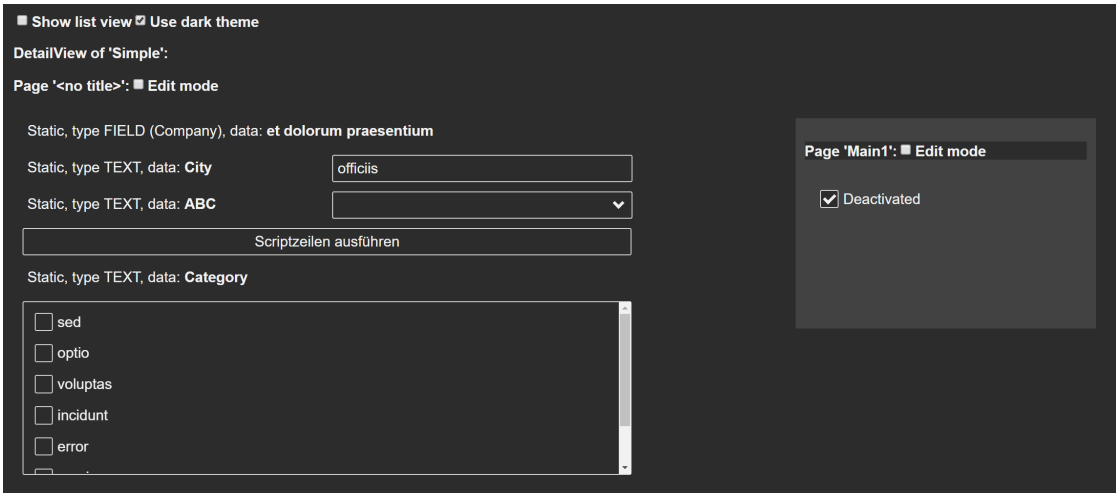


Abbildung 5.7: Detailansicht der Weboberfläche („Dark Mode“)

Sämtliche dargestellten Daten beider Ansichten, inklusive der generierten Dummy-Daten, wurden vom Mock-Server abgefragt, somit hat die Weboberfläche keinerlei Kenntnisse über die Daten. Die neuen Oberflächen enthalten noch einige Darstellungselemente, etwa die Checkboxen zum Wechseln des Modus von Übersichtsliste zur Detailansicht und Aktivieren des „Dark Mode“. Auch die Informationen zu den Elementtypen in der Detailansicht werden in der finalen Umsetzung nicht enthalten sein, durch die Anzeige von Mock-Daten ist es andernfalls jedoch schwierig zu erkennen, welches Daten zu welchem Feld gehören.

6

Fazit

Das ursprüngliche Ziel, die Umsetzung der gesamten UI, war in der zur Verfügung stehenden Zeit nicht realisierbar. Der Grund dafür ist, dass es zu viele UI-Elemente und notwendige Aspekte für deren Umsetzung gibt um diese in einer angemessenen Qualität bearbeiten zu können. Eine Modifikation der Ziele hin zur Erstellung eines Konzepts für die dynamischen Teile der *cRM* UI und das Erstellen eines simplen Prototypen war damit angebracht. Auf die Bearbeitung der statischen Elemente der UI, die Benutzerverwaltung, die Berichts- und Webansicht und die Termin- und Aufgabenplanung wurde somit verzichtet. Die modifizierten Ziele konnten erreicht werden.

Die Entwicklung mit *React* ist unkompliziert, die Auswahl an hilfreichen Bibliotheken groß und in der Community können zu jeder Fragestellung passende Antworten gefunden werden. *React* kann daher im folgenden Prozess weiterhin ohne Bedenken eingesetzt werden. Für ein qualifiziertes Fazit über die Nutzung von *GraphQL* reicht die bisher gesammelte Erfahrung nicht vollständig aus, der bisherige Eindruck ist aber positiv — es lässt sich jedoch festhalten, dass die Technologie sich im Rahmen dieses Projekts positiv bewährt hat. Eine weitere Evaluation von *GraphQL*, insbesondere auch in größeren Projekten, scheint somit empfehlenswert.

Insgesamt hat die Umsetzung des Konzepts gezeigt, dass es funktionieren kann. Der Ausbau ebendieses Konzepts ist jedoch unabdingbar, wobei es durch die in dieser Arbeit verzichteten Elemente weiter ergänzt werden sollte. Der Prototyp dient hierfür als solide Basis für die weitere Entwicklung. Zusätzliche Zeit wäre daher besonders wünschenswert gewesen, nicht zuletzt, um mehr Erfahrungen mit Anforderungsformulierungen und den Technologien zu sammeln sowie eine fundiertere Wissensbasis, insbesondere in den Bereichen des API-Designs und über *GraphQL*-Server, für die nächsten Schritte zu kumulieren.

6.1. Aufgetretene Schwierigkeiten

Wie oben erwähnt, wurde das ursprüngliche Ziel, die Umsetzung der gesamten UI, nicht erreicht. Obwohl es sich dabei nicht um eine technische Schwierigkeit handelte, hatte sie dennoch weitreichende Auswirkungen auf die letztendliche Umsetzung. Aufgrund dessen wurden zahlreiche zu Beginn der Arbeit antizipierten Schwierigkeiten, insbesondere die serverbasierte Persistenz von Einstellungen und die Entscheidung über die im Web umsetzbaren Features, nicht Gegenstand der Problemanalyse sowie der Problemlösung.

Des Weiteren wurde während der Evaluation der Frontend-Frameworks klar, dass es in diesem Bereich derart viele Ressourcen zu evaluieren gibt, dass eine vollständige Analyse dieser mehr als das Arbeitspensum eines einzelnen in Anspruch nehmen würde. Auf der anderen Seite sind sich die Technologien in ihrer Funktionalität aber dennoch so ähnlich, dass eine Entscheidung zum Großteil basierend auf persönlichen und betrieblichen Präferenzen getroffen werden musste. Trotz dessen eine möglichst objektive und zufriedenstellende Entscheidung für eine Technologie zu treffen, kostete mehr Zeit und Anstrengung als erwartet.

Eine weitere Schwierigkeit bestand darin, das gesetzte Ziel nicht aus den Augen zu verlieren. Ein Konzept für ein technisches Projekt umfasst zahlreiche Themengebiete und noch mehr Entscheidungen, welche sich zu einem unbestimmten Zeitpunkt in der Zukunft als Fehler erweisen könnten. Sämtliche Anforderungen müssen jederzeit bedacht werden und präsent sein. Zudem ist es notwendig Lösungen zu finden, welche all diesen Anforderungen gleichzeitig gerecht werden oder zumindest keine negativen Auswirkungen auf andere Bereiche nach sich ziehen. Diese mentale Belastung ist mitunter sehr anstrengend und kann bei der Entscheidungsfindung lähmen, wodurch die Erreichung des Ziels erschwert wird.

6.2. Ausblick

In diesem Abschnitt wird ein Ausblick auf zukünftige Entwicklungsmöglichkeiten zum weiteren Ausbau des Projektes gegeben. Zunächst sollen die noch nicht fertiggestellten Funktionalitäten abgeschlossen werden. Dazu zählen zwei *React*-Komponenten, ein Tab- und Container-Element, die Vervollständigung der Formatierungsmöglichkeiten und der Implementierung von Lade- und Fehlerzuständen aller Komponenten. Im Anschluss oder parallel kann weiterhin an einer verbesserten Umsetzung der *GraphQL*-API und auch der Abfragen geforscht werden. Die zu Beginn der Arbeit geringe

Erfahrung in diesem Bereich lässt darauf schließen, dass nicht alle Implementierungen optimal gelöst wurden und hier noch Verbesserungspotential besteht.

Sobald diese Aufgaben zufriedenstellend abgeschlossen sind, können die weiteren Aspekte der Anforderungsanalyse in Angriff genommen werden. Zum einen sind dies sämtliche weiteren Bestandteile der UI (inklusive der funktionalen Anforderungen an Web-Apps) und zum anderen das dafür benötigte Backend mit einem *GraphQL*-Server, der mit Echtdaten arbeitet.

Quellenverzeichnis

StackOverflow (März 2018a). *Stack Overflow Developer Survey 2018 - Company type*. Aufgerufen am 14.02.2019. URL: https://insights.stackoverflow.com/survey/2018/#company-type (siehe S. iii).

— (März 2018b). *Stack Overflow Developer Survey 2018 - Most popular technologies*. Aufgerufen am 14.02.2019. URL: https://insights.stackoverflow.com/survey/2018/#most-popular-technologies (siehe S. iii).

Bahr, Ines und Capterra (Jan. 2019). *Nutzerstudie: CRM Software Trends 2019 in deutschen KMU*. Aufgerufen am 14.02.2019. URL: https://www.capterra.com.de/blog/498/nutzerstudie-crm-software-trends-2019-in-deutschen-kmu (siehe S. iii).

GraphQL (o.D.). *Introduction to GraphQL*. Aufgerufen am 03.02.2019. URL: https://graphql.org/learn/ (siehe S. xi, 34).

Ishida, Richard, W3C, Susan K. Miller und Boeing (Feb. 2018). *Localization vs. Internationalization*. Aufgerufen am 03.02.2019. URL: https://www.w3.org/International/questions/qa-i18n (siehe S. 13).

Mehrabani, Afshin (o.D.). *Intro.js*. Aufgerufen am 15.03.2018. URL: https://introjs.com/ (siehe S. 14).

Greif, Sacha, Raphael Benitte und Michael Rambeau (Nov. 2018). *Front-end Frameworks - Conclusion*. Aufgerufen am 12.03.2019. URL: https://2018.stateofjs.com/front-end-frameworks/conclusion/quadrants (siehe S. 22).

Npmjs (Dez. 2018). *This year in JavaScript: 2018 in review and npm's predictions for 2019*. Aufgerufen am 07.03.2019. URL: https://blog.npmjs.org/post/180868064080/this-year-in-javascript-2018-in-review-and-npms (siehe S. 23, 33).

Weststrate, Michel (Feb. 2019). *mweststrate/immer*. Aufgerufen am 15.03.2018. URL: https://github.com/mweststrate/immer (siehe S. 43, 44).

Sturgeon, Phil (Juni 2017). *Representing State in REST and GraphQL*. Aufgerufen am 27.12.2018. URL: https://philsturgeon.uk/api/2017/06/19/representing-state-in-rest-and-graphql/ (siehe S. 47).

Apollo (Jan. 2019). *apollographql/graphql-tools*. Aufgerufen am 27.01.2019. URL: https://github.com/apollographql/graphql-tools (siehe S. 56).

GraphQL (Okt. 2018). *graphql/express-graphql*. Aufgerufen am 27.01.2019. URL: https://github.com/graphql/express-graphql (siehe S. 56).

Weiterführende Literatur

Fielding, Roy Thomas (2000). *Architectural Styles and the Design of Network-based Software Architectures*. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Wieruch, Robin (2018). *The Road to GraphQL: Your journey to master pragmatic GraphQL in JavaScript with React.js and Node.js*. Independently published.

Porcello, Eve und Alex Banks (2018). *Learning GraphQL: declarative data fetching for modern web apps*. OReilly.

Sturgeon, Phil (2015). *Build APIs you wont hate: everyone and their dog wants an API, so you should probably learn how to build one*. Createspace.

A

Anhang A

```
==================
QUERY
==================

{
  data {
    recordCount
    records {
      idField {
        ...RecordFieldFragment
      }
      fields(fieldNames: ["a", "b"]) {
        ...RecordFieldFragment
      }
    }
    record(id: "id1") {
      field(fieldName: "field1") {
        ...RecordFieldFragment
      }
      formulaContent(formula: "formula1")
    }
  }
}

fragment RecordFieldFragment on RecordField {
  name
  value
}

```

```
==================
RESPONSE
==================

{
  "data": {
    "recordCount": 3,
    "records": [
      {
        "idField": {
          "name": "id",
          "value": "ed5884be-9a31-4dfa-89e6-bbcf506a5ff4"
        },
        "fields": [
          {
            "name": "a",
            "value": "ut aliquid"
          },
          {
            "name": "b",
            "value": "magnam"
          }
        ]
      },
      {
        "idField": {
          "name": "id",
          "value": "613de566-6af3-49f2-b73c-3907ef90c04b"
        },
        "fields": [
          {
            "name": "a",
            "value": "maiores nam"
          },
          {
            "name": "b",
            "value": "dolorum et consectetur"
          }
        ]
      },
      {
        "idField": {
          "name": "id",
          "value": "fe3783a0-ba8d-4362-a1be-b8b3e91fa286"
        },
        "fields": [
          {
```

```
            "name": "a",
            "value": "doloremque distinctio maxime"
          },
          {
            "name": "b",
            "value": "perferendis dolores voluptatibus"
          }
        ]
      }
    ],
    "record": {
      "field": {
        "name": "field1",
        "value": "tempora cum aliquam"
      },
      "formulaContent": "Resolved formula: formula1"
    }
  }
}
```

Listing A.1: Abfrage und Antwort der Daten im *GraphQL*-Schema

```
==================
QUERY
==================

{
  listView {
    name
    columnLists {
      name
      columns {
        key
        value
      }
    }
  }
}

==================
RESPONSE
==================

{
  "listView": {
    "name": "Simple",
    "columnLists": [
```

```
26        {
27          "name": "Simple",
28          "columns": [
29            {
30              "key": "Company",
31              "value": "Company"
32            },
33            {
34              "key": "Country",
35              "value": "Country"
36            },
37            {
38              "key": "ZIP",
39              "value": "ZIP"
40            },
41            {
42              "key": "City",
43              "value": "City"
44            },
45            {
46              "key": "Street",
47              "value": "Street"
48            },
49            {
50              "key": "Phone",
51              "value": "Phone"
52            },
53            {
54              "key": "Email",
55              "value": "Email"
56            },
57            {
58              "key": "ModifiedOn",
59              "value": "ModifiedOn"
60            },
61            {
62              "key": "ModifiedBy",
63              "value": "ModifiedBy"
64            }
65          ]
66        }
67      ]
68    }
69  }
```

Listing A.2: Abfrage und Antwort der Listenansichtsstruktur im *GraphQL*-Schema

```
==================
QUERY
==================

{
  detailView {
    name
    mainPage {
      ...ControlsFragment
      pages {
        title
        ...ControlsFragment
        pages {
          title
        }
      }
    }
  }
}

fragment ControlsFragment on Page {
  controls {
    __typename
    ... on Static {
      text
      contentType
    }
    ... on Edit {
      field
    }
    ... on Button {
      caption
      bitmapIndex
      buttonAction
      parameter
    }
    ... on Listbox {
      field
    }
    ... on Combobox {
      field
      readOnly
    }
    ... on Checkbox {
      text
      field
```

```
    }
  }
}

==================
RESPONSE
==================

{
  "detailView": {
    "name": "Simple",
    "mainPage": {
      "controls": [
        {
          "__typename": "Static",
          "text": "Company",
          "contentType": "FIELD"
        },
        {
          "__typename": "Static",
          "text": "City",
          "contentType": "TEXT"
        },
        {
          "__typename": "Edit",
          "field": "City"
        },
        {
          "__typename": "Static",
          "text": "ABC",
          "contentType": "TEXT"
        },
        {
          "__typename": "Combobox",
          "field": "ABC",
          "readOnly": false
        },
        {
          "__typename": "Button",
          "caption": "Scriptzeilen ausführen",
          "bitmapIndex": "0",
          "buttonAction": "EXECUTE_SCRIPT",
          "parameter": "Testscript"
        },
        {
          "__typename": "Static",
          "text": "Category",
```

```
94              "contentType": "TEXT"
95            },
96            {
97              "__typename": "Listbox",
98              "field": "Category"
99            }
100         ],
101         "pages": [
102           {
103             "title": "Main1",
104             "controls": [
105               {
106                 "__typename": "Checkbox",
107                 "text": "Deactivated",
108                 "field": "Deactivated"
109               }
110             ],
111             "pages": []
112           }
113         ]
114       }
115     }
116 }
```

Listing A.3: Abfrage und Antwort der Detailansichtsstruktur im *GraphQL*-Schema

B

Anhang B

```
1  const InputValueHooks = () => {
2      const [editing, setEditing] = useState(false);
3      const [inputValue, setInputValue] = useState('');
4      const [value, setValue] = useState('');
5      const [borderColor, setBorderColor] = useState('blue');
6
7      const input = useRef(null);
8
9      const handleKeyPress = (e) => {
10         if (e.key === 'Enter') {
11             this.setState({value: this.state.inputValue})
12         } else if (e.key === 'Escape') {
13             this.input.current.value = this.state.value;
14             this.setState({inputValue: this.state.value})
15         }
16     }
17
18     return (
19         <div>
20             <input
21                 onFocus={() => {setEditing(true); setBorderColor('green')
                       ;}}
22                 onBlur=={() => {setEditing(false); setBorderColor('blue')
                       ;}}
23                 type="text"
24                 className={`inputForm ${editing ? 'editBorder' : '
                       normalBorder'}`}
25                 onChange={(event) => {setInputValue(event.target.value);}}
26                 onKeyDown={handleKeyPress}
```

```
27                  ref={input}
28              />
29              {editing ?
30                  <span>
31                  <span className="submitInfo">Press enter to save changes,
                        </span>
32                  <span className="abortInfo">press esc to discard changes</
                        span>
33                  </span> : null
34              }
35              <div>
36                  Current value: {value}
37              </div>
38          </div>
39      )
40  }
```

Listing B.1: Funktionale Umsetzung einer Beispielkomponente mit *React*

C

Anhang C

```
1  public void Visit(DialogElement dialog)
2  {
3      foreach (var page in dialog.Pages)
4      {
5          page.Apply(this);
6      }
7  }
8
9  public void Visit(PageElement page)
10 {
11     if (_currentPage != null)
12     {
13         throw new InvalidOperationException("Page must be processed
               completely before continuing to next page");
14     }
15
16     _currentPage = new Page(page.Name);
17     _currentPage.AddMetaData(page.MetaData);
18
19     page.ControlList.Apply(this);
20
21     _dli.Add(_currentPage);
22     _currentPage = null;
23 }
24
25 public void Visit(ControlListElement controlList)
26 {
27     if (_currentPage == null)
28     {
```

```
29          throw new InvalidOperationException("Page must be valid to process
                its control list");
30      }
31
32      foreach (var control in controlList.Controls)
33      {
34          control.Apply(this);
35      }
36  }
37
38  public void Visit(ControlElement control)
39  {
40      if (_currentPage == null)
41      {
42          throw new InvalidOperationException("Page must be valid to process
                its controls");
43      }
44
45      _currentPage.AddControl(CreateControlByType(control.Type, control.
            Items));
46  }
47
48  private IJsonConvertible CreateControlByType(string type, Dictionary<
        string, string> items)
49  {
50      switch (type)
51      {
52          case "STATIC":
53              return StaticControl.ConstructWithItems(items);
54          case "BUTTON":
55              return ButtonControl.ConstructWithItems(items);
56          case "EDIT":
57              return EditControl.ConstructWithItems(items);
58          default:
59              return null;
60      }
61  }
```

Listing C.1: Ablaufen und Extrahieren relevanter Informationen aus XML-Tokens durch den JSON-Visitor