

B

C

Bachelorthesis

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

S

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 [StackOverflow 2018a] und arbeiten respektive mit entsprechenden Technologien [StackOverflow 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 (ThinClient¹) 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 Nutzer² 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.

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

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

Inhaltsverzeichnis

Abstract	iii
Abbildungsverzeichnis	vi
Quellcodeverzeichnis	vii
Abkürzungsverzeichnis	xi
1 Einleitung	1
1.1 Vorgehen und Gliederung	1
1.2 Motivation	1
1.3 Erwartete Schwierigkeiten	2
2 Anforderungsanalyse	5
2.1 Produktüberblick	5
2.2 Funktionale Anforderungen	7
2.2.1 combit Relationship Manager	8
2.2.2 Webbereich	12
2.3 Anforderungen an den Entwicklungsprozess	16
2.4 Hauptaugenmerk dieser Arbeit	17
3 Technologien	19
3.1 Entwicklungsumgebung	19
3.2 Frontend-Frameworks	20
3.2.1 Traditionelle Lösungen	20
3.2.2 Moderne, JavaScript-basierte Lösungen	20
3.2.3 Vergleich Vue, React & Angular	21
3.2.4 Fazit.	29
3.3 TypeScript	30
3.4 API-Anbindung	31
3.4.1 Rest.	31
3.4.2 GraphQL	32
3.4.3 Fazit.	34

4	Ausarbeitung des Architekturkonzepts	35
4.1	Analyse und Übersetzung der bisherigen UI-Struktur	35
4.1.1	Detailansicht (DLI)	35
4.1.2	Übersichtsliste (VLC)	36
4.1.3	Neue Struktur	36
4.2	Webseite	37
4.2.1	React-Komponenten	37
4.2.2	Komponenten-Layout	37
4.2.3	Identifikation auf Server	39
4.2.4	Visualisierung von Lade- und Fehlerzustände	39
4.2.5	Einbindung von GraphQL mit Apollo	40
4.2.6	Unveränderbare Daten	41
4.2.7	Editier-Modus	41
4.2.8	Individualisierung	42
4.2.9	Suche, Filter, Sortierung	42
4.2.10	Tests und Continuous Integration	42
4.3	API	43
4.3.1	Aufbau / GraphQL-Schema	43
4.3.2	Keine Business Logik im Client	43
5	Implementierung eines Prototypen	45
5.1	Tool zum Parsen der vorhandenen UI-Struktur	45
5.2	Webseite	50
5.2.1	React-Komponenten	50
5.2.2	Tests	52
5.2.3	GraphQL-Mock-Server und Resolver	52
5.3	Beispielumsetzung	54
6	Fazit	55
6.1	Aufgetretene Schwierigkeiten	55
6.2	Ausblick	56
	Quellenverzeichnis	57

Abbildungsverzeichnis

2.1	Grafische Oberfläche des cRM Desktopclient (Februar 2019)	6
2.2	Aktueller technischer Aufbau des cRM	7
2.3	Möglicher, zukünftiger technischer Aufbau des cRM	8
2.4	Anleitungs-Modus von Intro.js [Mehrabani o.D.]	13
3.1	Beispielkomponente für die Umsetzung mit allen Frameworks	22
4.1	Eigener Layout-Prototyp mit CSS-Grids	38
4.2	Ladezustand einer Komponente	40
4.3	Mögliche Fehlerzustände einer Komponente	40
4.4	Editier-Konzept von immer.js [Weststrate 2019]	41
5.1	Datei-Struktur des Konvertierungstools	46
5.2	Klassendiagramm der Visitor-Struktur des Konvertierungstools	49
5.3	Beispiel für die Komponentendarstellung mit Storybook	53

Quellcodeverzeichnis

3.1	Umsetzung der Beispielkomponente mit Vue	22
3.2	Umsetzung der Beispielkomponente mit React	23
3.3	Funktionale Umsetzung der Beispielkomponente mit React	25
3.4	Umsetzung der Beispielkomponente mit Angular	26
3.5	Beispielquery aus der GraphQL Dokumentation [GraphQL o.D.]	33
4.1	Datentypen der neuen Detailansicht-Struktur	35
4.2	Datentypen der neuen Übersichtslisten-Struktur	36
5.1	Initialisierung der PageElement-Klasse	46
5.2	Apply-Methode der DialogElement-Klasse	47
5.3	Ablaufen von und Extrahieren relevanter Informationen aus XML-Tokens durch den JSON-Visitor	47
5.4	XML-Input	49
5.5	JSON-Ergebnis	50
5.6	Benutzung der styled-components Bibliothek	51
5.7	Einbettung von GraphQL-Query-Fragmenten	51
5.8	Snapshot-Test mit Jest und Enzyme	52
5.9	Konfiguration einer Storybook Komponenten	52
5.10	Teil der GraphQL-Schemadefinition	53
5.11	GraphQL Schema-Resolver	53

Abkürzungsverzeichnis

Bezeichnung	Beschreibung	Seiten
CLI	Command-Line-Interface	21, 42
COM	Component Object Model	10, 11
CRA	create-react-app	42
CRM	Customer Relationship Management	iii, 5
cRM	combit Relationship Manager	iii, v, vii, 2, 5–12, 14, 17, 19, 37, 54, 55
DSGVO	Datenschutz-Grundverordnung	5, 15
LL	List & Label	9
SSR	Server-Side-Rendering	29
WYSIWYG	What You See Is What You Get	9

1

Einleitung

1.1. Vorgehen und Gliederung

Kapitel 1 führt in die Thematik ein, erläutert die Motivation der Arbeit und gibt einen Überblick über im Voraus antizipierte Schwierigkeiten bezüglich der Umsetzung des Architekturkonzepts. In Kapitel 2 werden Anforderungen (auch anhand bisheriger Features) ermittelt. In Kapitel 3 werden die in Betracht gezogenen Technologien und deren Zusammenspiel untereinander vorgestellt. Dazu gehören die zu nutzende Programmierumgebung (Sprache, Testframework, Continuous Integration), der Vergleich verschiedener Frontend-Frameworks, die anhand der in Kapitel 2 formulierten Anforderungen bewertet werden und die Protokolle zur Kommunikation mit dem Backend. In Kapitel 4 wird die Ausarbeitung des Konzepts mit den in Kapitel 3 ausgesuchten Technologien und der Aufbau der API zur Kommunikation mit dem Backend beschrieben. In Kapitel 5 wird der Entwicklungsprozess der beispielhaften Umsetzung des erstellten Konzepts dargestellt. In Kapitel 6 wird das Ergebnis kritisch hinterfragt, die tatsächlich aufgetretenen Schwierigkeiten mit den zuvor Erwarteten verglichen und ein Ausblick auf weitere Entwicklungsschritte gegeben.

1.2. Motivation

FIXME: Problemstellung und Zielsetzung? 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

FIXME: transitions between sections

2.1. 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) durch ein zweites internes Produkt "List & Label". Diese bauen jedoch auf den Daten einer Lösung auf. Eine *Solution* stellt die grundsätzliche Organisation der Daten in bestimmte



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

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 verwalten-den 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

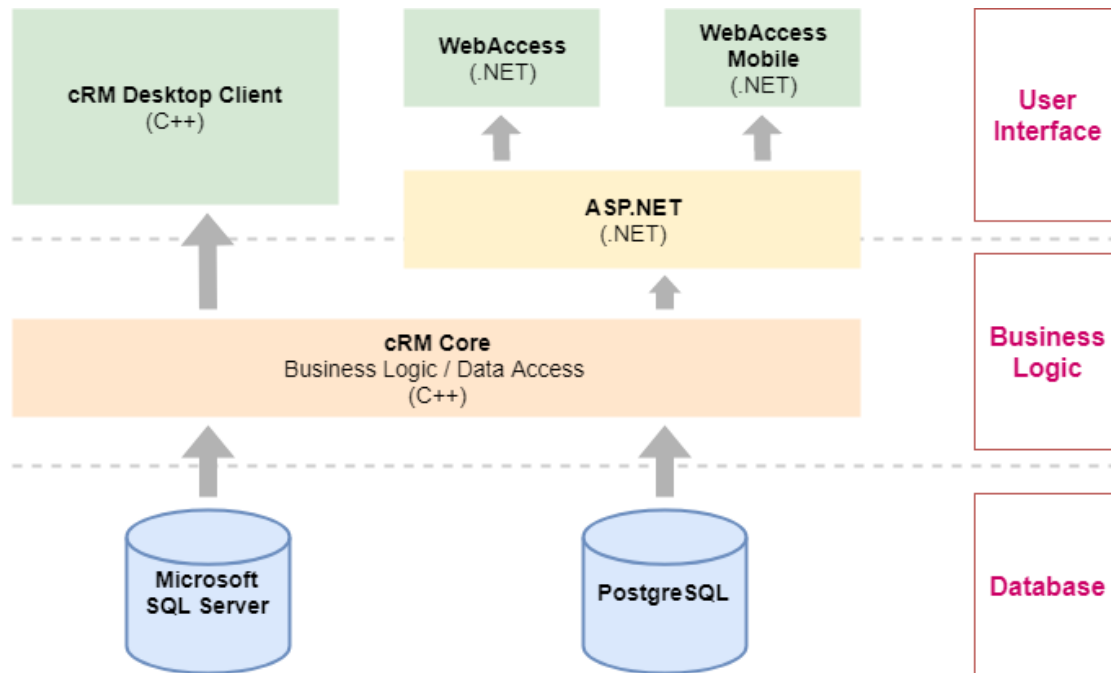


Abbildung 2.2: Aktueller technischer Aufbau des cRM

den Datenbankzugriff und die Business Logic¹ 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.NET² 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.

2.2. Funktionale Anforderungen

In diesem Teil des Kapitels werden funktionale Anforderungen, welche die Web-UI unterstützen soll, aufgelistet und kurz erläutert. Der erste Abschnitt befasst sich mit Anfor-

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

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

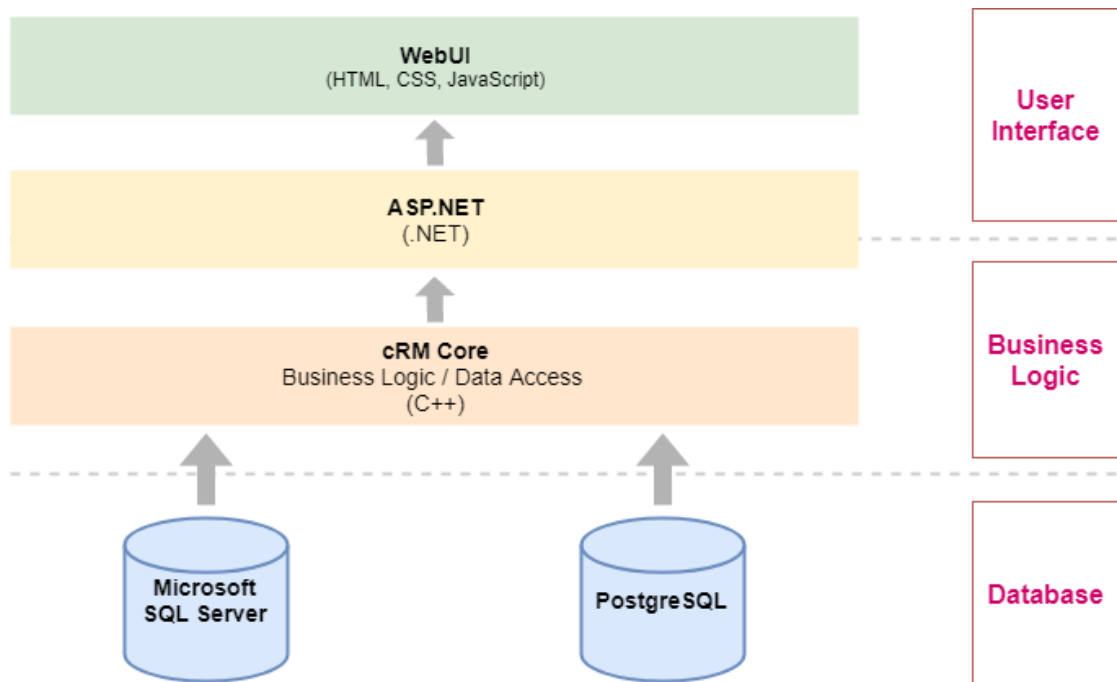


Abbildung 2.3: Möglicher, zukünftiger technischer Aufbau des cRM

derungen, die sich aus dem cRM ergeben, der zweite Abschnitt mit Anforderungen, die sich aus Web-Apps ergeben.

2.2.1. combit Relationship Manager

Automatisierte Erstellung

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.

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.

Edit-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.

Individualisierung

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)"-Prinzip¹ das Designen erleichtert. Die Weboberfläche sollte diese Funktionalität möglichst vollständig abbilden.

Berichts- und Webansicht

Wie in Abschnitt 2.1 erwähnt, werden Berichte von einem zweiten Produkt der Firma *combit GmbH*, dem sogenannten *List & Label (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`-Tag² eingebunden werden.

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

¹Echtzeitdarstellung des Ergebnisses während der Erstellung und Bearbeitung

²HTML-Element, welches das Einbetten von Dokumenten in andere Dokumente erlaubt

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.

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.

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).

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.

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 werden, da dies weitere Downloads von irrelevanten Daten (CSS für nicht-mobile Geräte) vermeidet. Für stationäre Geräte ist eine derartige Optimierung jedoch nicht vonnöten.

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:

FIXME: itemize alternative?

- 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, Interaktion 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.)

2.2.2. Webbereich

Zusätzlich zu diesen 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 wichtigsten dieser Anforderungen werden im Folgenden kurz erläutert.

i18n / i10n

Die Akronyme “i18n” und “i10n” stehen für Internationalisierung respektive Lokalisierung [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.

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

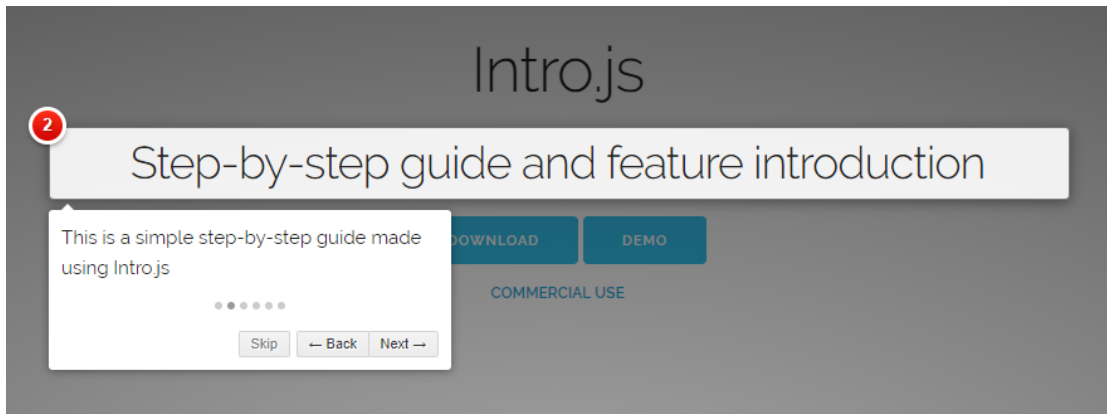


Abbildung 2.4: Anleitungs-Modus von Intro.js [Mehrabani o.D.]

und wie diese bei der Softwareentwicklung beachtet und umgesetzt werden können.

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.

“Offline-First“-Ansatz

In jüngster Zeit ist es möglich, mithilfe von Service-Workern¹ 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, 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.

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

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.

Etablierte Authentifizierung

Der *cRM* besitzt zwar bereits ein integriertes Benutzersystem mit 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.

Caching

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 gespeicher-

ten 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.

Container

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.

Serverless

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.

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.

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.

2.3. 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, folgende Technologien und Praktiken bei der Entwicklung des neuen Projektes zu nutzen:

FIXME: itemize alternative?

- 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.

- Continuous 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.

- 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.

2.4. 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 folgende Punkte beachtet:

- Automatisierte Erstellung
- Editier-Modus
- Individualisierung
- Suche / Filter / Sortierung

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 transpiliert¹ 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.

¹Übersetzung von Quellcode aus einer Programmiersprache in eine andere

3

Technologien

Die richtige Technologie zur Lösung einer Problemstellung auszuwählen garantiert noch keinen Erfolg, ist aber ein wichtiger Grundstein, um überhaupt erfolgreich sein zu können. In diesem Sinne sollte nicht zu wenig Augenmerk auf den Vergleich und die Entscheidung zwischen auf dem Markt verfügbaren Alternativen gelegt werden. 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 momentan ausschließlich mit *Microsoft Visual Studio* statt. Wie in Kapitel 2 beschrieben ist, gibt es die Core-Komponente und Desktopkomponente, welche in C++ geschrieben sind, und eine auf den Core aufsetzende, in C# geschrieben, .NET-Komponente zur Bereitstellung der Webfunktionalität. Der bestehende Entwicklungsprozess soll nicht verändert und auch nicht weiter thematisiert werden, da die neu zu entwickelnde UI-Schicht eine unabhängige Codebasis besitzt und auch konzeptionell von den bisherigen Schichten getrennt ist.

Für das in Kapitel 2 angesprochene Tool zum Einlesen und Konvertieren der bisherigen UI-Repräsentation wurde wegen der vorhandenen internen Infrastruktur ebenfalls Visual Studio verwendet. Die Wahl der Programmiersprache fiel auf C#, da der Zugriff auf das Dateisystem und speziell auf XML-Dateien, das Format der momentanen Repräsentation, sehr einfach ist.

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 die einzige Sprache, die von allen Browsern unterstützt wird. Das spiegelt sich auch in der Auswahl der Frontend-Frameworks wieder, welche alle entweder direkt oder indirekt in JavaScript geschrieben sind.

3.2. Frontend-Frameworks

3.2.1. Traditionelle Lösungen

Frameworks und Bibliotheken, die versuchen, Entwicklern das Erstellen von UIs zu erleichtern, gibt es schon sehr lange 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 pures 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 um komplette Webseiten von Grund auf zu erstellen — vielmehr können sie mit den unten angesprochenen JavaScript-Projekten im Zusammenspiel benutzt werden.

3.2.2. Moderne, JavaScript-basierte Lösungen

In diesem Bereich sind vor allem komponentenbasierte Frameworks, basierend auf einem virtuellen DOM¹, stark verbreitet. Aufgrund der zeitlichen Limitierung dieser Arbeit ist es nicht möglich, die große Auswahl an Alternativen zu betrachten. 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.

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

3.2.3. Vergleich Vue, React & Angular

Generell

- Vue

Die Entwickler legen viel Wert darauf eine möglichst kleine und dennoch performante Laufzeit zu entwickeln. Der Einstieg in *Vue* soll möglichst einfach sein, was durch eine sehr gute Dokumentation und ein hervorragendes Command-Line-Interface (CLI)-Tool zum Erstellen neuer Projekte gelingt. Abhängigkeiten zwischen Daten und UI verfolgt *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, die Entwickler arbeiten mit diesen aber zusammen um den Prozess zu verbessern. Das Verwenden von JavaScript-Templates (JSX) und *Typescript* ist mit etwas Konfiguration ebenfalls möglich. *Vue* hat im Vergleich noch ein kleines Ökosystem, kann aber dafür den stärksten Wachstum aller Frameworks verzeichnen [vgl. Npmjs 2018].

- React

React diktiert im Vergleich zu *Vue* noch weniger Vorgaben an die Entwickler, alle typischen Problemstellungen werden durch Bibliotheken gelöst. Dies ist möglich, da das Ökosystem sehr groß ist und eine breite und tiefe Auswahl an entsprechenden Bibliotheken bietet. Native Apps (Android, iOS) können mit dem vom gleichen Team entwickelten *React-Native* erstellt werden, ohne dafür weitere Software zu benötigen. Durch die Konzentration auf wesentliche Features und Verbesserungen und 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, Type checks, Autocompletion) sind problemlos möglich.

- Angular

Es werden für alle typischen Anforderungen die bei der Entwicklung anfallen bereits Tools und Konzepte mit ausgeliefert, um diese Anforderungen zu erfüllen. 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. Der Einstieg in *Angular* wird von vielen daher als mühsam und zeitintensiv beschrieben. Die Dokumentation ist zwar sehr umfangreich, teilweise aber auch unüber-

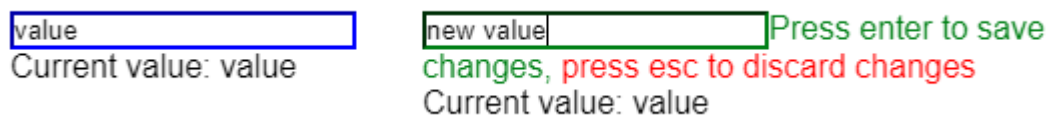


Abbildung 3.1: Beispielkomponente für die Umsetzung mit allen Frameworks

sichtlich. Das Ökosystem von *Angular* ist noch größer als das von *Vue*, es hat jedoch auch negatives Wachstum zu verzeichnen [vgl. Npmjs 2018].

Aufbau von Apps (mit Beispielkomponente)

Um zumindest ein Gefühl für das Benutzen der jeweiligen Frameworks zu bekommen wurde eine Beispielkomponente ausgewählt die mit allen drei Frameworks umgesetzt werden soll. In Abbildung 3.1 ist die fertige Komponente dargestellt. Bei Klick in das Edit-Feld ändert sich die Farbe des Rahmens um den Fokus 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 fertig Umgebung mit dem jeweiligen Framework bieten. Auf die Benutzung weiterer Bibliotheken wurde hier explizit verzichtet.

- Vue

Die *Vue*-Komponente wird komplett als Instanz einer *Vue*-Klasse angelegt, der Daten und Methoden übergeben werden müssen. Im HTML-Template werden die *Vue*-eigenen Attribute ersichtlich.

```

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
40             save changes, </span>
41         <span v-if="editing" class="abortInfo">press esc to
42             discard changes</span>
43         <div>
44             Current value: {{value}}
45         </div>
46     </div>
47 `
48 },
49 });

```

Listing 3.1: Umsetzung der Beispielkomponente mit Vue

- React

Eine *React*-Komponente kann entweder als herkömmliche Klasse oder nur als Funktion, welche die darzustellenden UI-Elemente als Rückgabewert enthält, angegeben werden.

```

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
11      this.input = React.createRef();
12
13      this.onFocus = this.onFocus.bind(this);
14      this.onFocusLost = this.onFocusLost.bind(this);
15      this.handleKeyPress = this.handleKeyPress.bind(this);
16      this.onChange = this.onChange.bind(this);
17    }
18
19    onFocus() {
20      this.setState({editing: true, borderColor: 'green'});
21    }
22
23    onFocusLost() {
24      this.setState({editing: false, borderColor: 'blue'});
25    }
26
27    handleKeyPress(e) {
28      if (e.key === 'Enter') {
29        this.setState({value: this.state.inputValue})
30      } else if (e.key === 'Escape') {
31        this.input.current.value = this.state.value;
32        this.setState({inputValue: this.state.value})
33      }
34    }
35
36    onChange(event) {
37      this.setState({inputValue: event.target.value})
38    }
39
40    render() {
41      return (
42        <div>
43          <input
44            onFocus={this.onFocus}
45            onBlur={this.onFocusLost}
46            type="text"
47            className={`inputForm ${this.state.editing ? '
              editBorder' : 'normalBorder'}`}
48            onChange={this.onChange}
49            onKeyDown={this.handleKeyPress}
50            ref={this.input}
51          />
52          {this.state.editing ?
53            <span>
54              <span className="submitInfo">Press enter to save

```

```

        changes, </span>
55      <span className="abortInfo">press esc to discard
        changes</span>
56    </span> : null
57  }
58  <div>
59    Current value: {this.state.value}
60  </div>
61 </div>
62 )
63 }
64 }

```

Listing 3.2: Umsetzung der Beispielkomponente mit React

```

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('
22           green')}}
23         onBlur={() => {setEditing(false); setBorderColor('
24           blue')}}
25         type="text"
26         className={`inputForm ${editing ? 'editBorder' : '
27           normalBorder'}}`
28         onChange={(event) => {setInputValue(event.target.
29           value)}}
30         onKeyDown={handleKeyPress}
31         ref={input}
32       />

```

```

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 3.3: Funktionale Umsetzung der Beispielkomponente mit React

- Angular

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.

```

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 3.4: Umsetzung der Beispielkomponente mit Angular

State-Management

- Vue

Standard ist, dass jede Komponente ihr eigenes State-Objekt besitzt. Zusätzlich dazu kann noch auf das State-Objekt der *Vue*-Instanz, in der sich die Komponente befindet, zugegriffen werden. Dieses kann entweder direkt verändert, oder per Store-Pattern¹ verwaltet werden. *Vuex* ist eine direkt von *Vue* entwickelte Bibliothek, welche das genannte Store-Pattern umsetzt.

- React

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

- Angular

¹Geteilter State wird zentral verwaltet, Änderungen am State nur intern möglich

²unveränderbare Übergabeparameter

State wird 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 dem gleichen Prinzip wie *Redux* beruht, aufgrund moderner *Typescript*-Features laut *Angular* aber einfacher zu benutzen sei.

Routing

- Vue

Routing ist direkt in *Vue* integriert. Routen werden in JavaScript mit Komponenten verknüpft und bei Aufruf dieser Route wird die verknüpfte Komponente an einen zuvor festgelegten Platzhalter im DOM gerendert.

- React

Nicht integriert, es gibt aber mehrere Alternativen als Bibliotheken. Bei *Aviator* werden Routen als geschachteltes Objekt übergeben, bei dem jedes Element eine Route und eine Zielfunktion enthält. Die Zielfunktion wird aufgerufen, wenn zur entsprechenden Route navigiert wird und muss wissen, 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, und die eigentliche Komponente, die angezeigt werden soll.

- Angular

Analog zu *Vue* ist auch hier eine Lösung direkt integriert und funktioniert auch nach dem exakt gleichen Prinzip.

Testing

- Vue

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.

- React

Das Testen von *React*-Komponenten funktioniert analog zu *Vue*.

- Angular

Alle zum Testen benötigten Tools sind 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 durch die Notwendigkeit, sich mit dem kompletten Test-Konzept auseinandersetzen zu müssen, höher als bei *Vue* und *React*.

Server-Side-Rendering

Bei allen Frameworks gibt es auch die Möglichkeit nur wenige Seiten “vor-gerendert” auf dem Server liegen zu haben, anstatt sie dynamisch bei entsprechenden Anfragen zu generieren. Hierfür kann das *Node.js* Tool *Prerender.io* benutzt werden.

- Vue

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

- React

Auch *React* liefert einen dafür vorgesehenen Renderer (*ReactDOMServer*), der Markup als Text generiert. Das kann beim initialen Laden der Seite ausgeliefert und auf Clientseite mit *ReactDOM.hydrate()* mit Inhalten gefüllt werden (erfordert einen *Node.js* Server).

- Angular

Angular benötigt für Server-Side-Rendering (SSR) zusätzliche Abhängigkeiten, Änderungen an der Konfiguration, ein zusätzliches Build / Bundle Target und ein zusätzliches Modul mit extra Konfiguration, das auf dem Server läuft und dafür zuständig ist das neue JavaScript-Bundle auszuliefern.

3.2.4. Fazit

Alle betrachteten Alternativen sind valide Optionen, mit denen jegliche Anforderungen umgesetzt werden können. Der Einstieg in *Angular* scheint allerdings schwierig, da dort eine Vielzahl an Technologien für verschiedenste Zwecke mitgeliefert werden. Da es seitens der Funktionalität keine Unterschiede gibt, kommt es mehr oder weniger darauf an, welches Projekt einem Entwickler (-team) am meisten zusagt bzw. mit welchem es am besten arbeiten kann. *React* besitzt mit Abstand die größte Community, es kann daher davon ausgegangen werden, dass die Weiterentwicklung, viele 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 also am Entwickler das entsprechende Wissen aufzubauen und keine Fehler zu machen.

Für den Prototypen wird vorerst *React* genutzt. Da die Technologien zur Laufzeit alle kompatibel sind (sie werden alle in normales HTML, CSS und JavaScript transpiliert), kann dieser Code später aber auch mit *Vue* zusammen genutzt oder nach und nach 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 schnell kleinere Skripte geschrieben werden, bei denen aufgrund ihrer überschaubaren Größe entsprechende Überprüfungen trivial sind, oder es schlichtweg keine Rolle spielt, ob ein Skript Fehler enthält. 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 (normaler JavaScript-Code ist also ebenso gültiger *TypeScript*-Code) beschrieben werden kann, zu gewöhnlichem JavaScript transpiliert wird und somit trotz ihrer Vorteile keinen Kompatibilitätsnachteil besitzt. Da die Syntax auf JavaScript basiert, ist das Erlernen dieser für JavaScript-Entwickler trivial. Durch Nutzung von typisierten Daten werden fehlerhafte Nutzungen von 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 das Anbieten von Codevervollständigung, was es Ent-

wicklern oft erspart die Dokumentation konsultieren zu müssen und damit zu einem effektiveren Entwicklungsprozess beiträgt.

Der einzige Nachteil von *TypeScript* ist, dass ein zusätzlicher Schritt zum Übersetzen in JavaScript-Code notwendig ist. Da die meisten Projekte im Webbereich aber sowieso nicht komplett ohne ähnliche Tooling-Schritte auskommen und der *TypeScript*-Compiler einfach in diese bestehenden Prozesse integrierbar ist, handelt es sich hierbei nur um eine überschaubare Einschränkung. Dieser Nachteil kann aber auch positiv ausgelegt werden, da es durch den Übersetzungsschritt möglich ist, 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 diesem Abschnitt werden speziell die für dieses Projekt relevanten Vor- und Nachteile beider Ansätze angesprochen.

3.4.1. Rest

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 State in den nächsten sind bereits in der momentanen Antwort einer Anfrage in Form von Links und Metadaten verfügbar. Ein Konsument muss damit nicht mehr wissen, welche Anfragen zu welchem

¹Hypermedia as the Engine of Application State

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 von Clients und Server) entfällt ebenfalls, neue Features können als weitere Aktion-Links auftauchen und alte Features zusätzlich zu einem Link die Information erhalten, dass die Aktion bald nicht mehr zur Verfügung stehen wird. Trotz verschiedener Standards die beim Erstellen der Struktur der Daten, vor allem bezgl. der Metadaten und wie diese ausgelesen werden können, 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 gemacht 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 möglich, indem dauerhaft die Dokumentation (welche in entsprechender Qualität vorhanden sein muss) konsultiert wird. Bei Änderungen an der API müssen auch die Clients angepasst werden, da sie keine Möglichkeit haben die Änderungen direkt mitzubekommen. Um die Erstellung von richtigen *Rest-APIs* zu vereinfachen, 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*, da jedes Endpoint-Daten-Paar als ein Eintrag im Cache angesehen werden kann. 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 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 granularer diese Einschränkungen sind, desto weniger überflüssige Daten müssen übertragen werden, gleichzeitig greift damit aber auch der Cache immer seltener, weil dauerhaft verschiedene Endpoint-Daten-Paare angefragt werden.

3.4.2. GraphQL

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 ist deswegen, in diesem Bereich möglichst sparsam zu sein und möglichst wenig Daten zu übertragen. *GraphQL* ist laut *npm* die im Webbereich mit am meisten wachsende Technologie überhaupt ("hyper-growth" [vgl. [Npmjs 2018](#), unteres Drittel]), als Folge dessen wird es in Zukunft wahrscheinlich in vielen Pro-

jekten benutzt werden, der Aufbau von entsprechendem Know-How in diesem Bereich ist daher wichtig. Eine Abfrage bei *GraphQL* hat den gleichen Aufbau wie ein JSON-Dokument, mit der Besonderheit, dass anstatt Schlüssel-Werte-Paaren nur Schlüssel bzw. der Name von Ressourcen angegeben werden. Eine Antwort hat immer den gleichen Aufbau wie eine Anfrage, die Schlüssel bzw. Ressourcennamen werden dabei, wie in Listing 3.5 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 Einsprungpunkte eines Graphen, kann er mit diesen Metainformationen alle weiteren Informationen programmatisch auslesen.

```
1 Query:
2 {
3   me {
4     name
5   }
6 }
7
8 Result:
9 {
10  "me": {
11    "name": "Luke Skywalker"
12  }
13 }
```

Listing 3.5: Beispielquery aus der GraphQL Dokumentation [GraphQL [o.D.](#)]

Um Entwickler zu unterstützen, existiert ein visueller Query-Editor *GraphiQL* mit Autovervollständigung und aus dem Schema generierter Dokumentation. Mithilfe dieses Editors kann sehr leicht in der API navigiert und 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, deren Änderung er sofort mitbekommen 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 diese ganzen Eigenschaften zu ermöglichen ist es jedoch notwendig, dass sowohl der Client als auch der Server mit dem gleichen *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 das Erstellen von Queries, Caching und Debuggen erleichtern, sind *Apollo* und *Relay*. *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 für den gleichen Zweck benutzt werden, sie konkurrieren jedoch nicht direkt miteinander. Eine API kann ebenso beide Ansätze entweder ergänzend oder parallel anbieten. Bestehende *Rest*-APIs können durch das Resolver-Konzept von *GraphQL* auf einfache Art und Weise umhüllt werden und so auf beide Arten angesprochen werden. *Rest* ist flexibler als *GraphQL*, um diese Flexibilität aber richtig zu nutzen und damit eine API zu schreiben, welche viele Jahre genutzt und skaliert werden kann, ist aber viel Erfahrung und Aufwand notwendig. *GraphQL* scheint daher als die richtige Wahl für Firmen oder Personen, die noch nicht viel Erfahrung in diesem Bereich sammeln konnten. *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, typsichere Anfragen zu erstellen. Ebenso wird mit dem mitgelieferten *GraphiQL*-Tool ein Abfrage-Editor mit Autovervollständigung und Fehlerbeschreibungen bereitgestellt, der es Nutzern sehr einfach macht eine API und deren Möglichkeiten zu erkunden sowie für alle Bestandteile direkt eine Dokumentation einzusehen. 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 Echt Daten 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

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 leider keine offizielle Spezifikation und auch keine Dokumentation, es ist daher eine Herausforderung direkt alle nötigen Informationen aus ihnen auszulesen und bedarf unter Umständen spätere Anpassungen am Auslesetool oder der künftigen Struktur. In diesem Abschnitt werden nur die tatsächlich benötigten bzw. ausgelesenen Informationen beschrieben.

4.1.1. Detailansicht (DLI)

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 Name, einer Hintergrundfarbe, etc., einem **Controls**-Element und den darin enthaltenen **Control**-Elementen. Ein **Control**-Element hat einen bestimmten Typ und enthält abhängig von diesem etliche weitere Eigenschaften die typspezifisch ausgewertet werden müssen. Die grundsätzliche Form der neuen Struktur ist in Auflistung 4.1 beschrieben.

```
2     "view": String,  
3     "pages": Array<{  
4         "name": String,  
5         "controls": Array<{  
6             "type": String, // Static, Edit, Button, ...  
7             "database_field": String,  
8             ... // Typspezifische Felder  
9         }>  
10    }>  
11 }
```

Listing 4.1: Datentypen der neuen Detailansicht-Struktur

4.1.2. Übersichtsliste (VLC)

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.

```
1 {  
2     "view": String,  
3     "lists": Array<{  
4         "database_table": String,  
5         "columns": Array<{  
6             "column": String,  
7             "display_text": String  
8         }>  
9     }>  
10 }
```

Listing 4.2: Datentypen der neuen Übersichtslisten-Struktur

4.1.3. Neue 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 denen 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’ (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. 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 die Darstellungsstruktur vom Server abgefragt und mit dieser Antwort vorgefertigte UI-Komponenten dynamisch platziert. Bei externen Änderungen der Daten wird dieser Vorgang mithilfe von GraphQL-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 Darstellungs-Elementen 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 aus dem Datenbankfeld das mit dem jeweiligen Element verknüpft ist und 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.

4.2.2. Komponenten-Layout

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

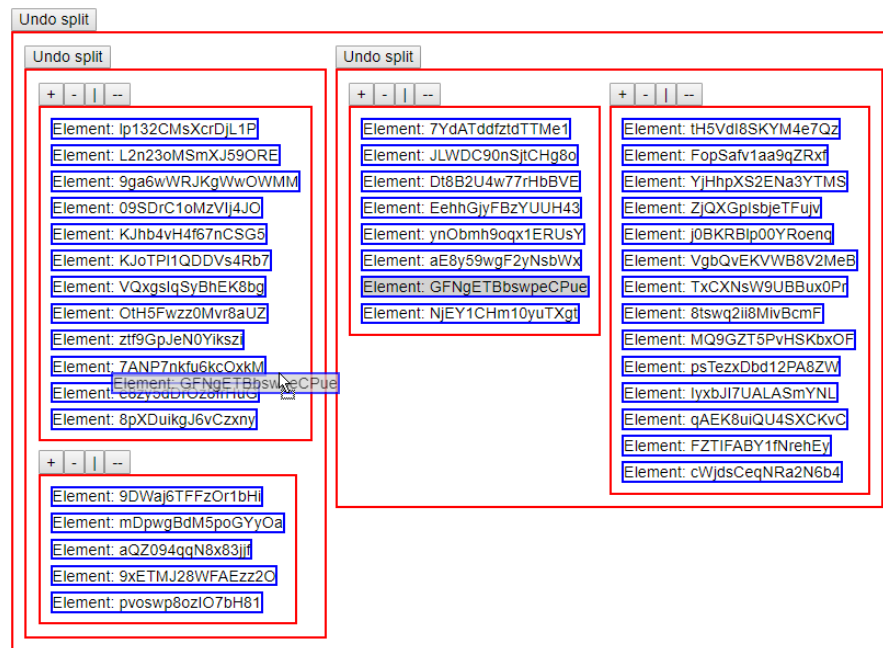


Abbildung 4.1: Eigener Layout-Prototyp mit CSS-Grids

Erste Überlegungen

Zu Beginn erschien die zentrale Frage, wie es technisch möglich ist, die Komponenten der Detailansicht vom Benutzer anpassbar anzuordnen. Eine naive Herangehensweise kann in Abbildung 4.1 gesehen werden. Umgesetzt wurde dies mit einem CSS-Grid (rote Umrandung) das entweder horizontal oder vertikal in zwei Hälften getrennt werden kann. Jede dieser Hälften stellt abermals ein CSS-Grid dar das beliebig zwischen Elterncontainern verschoben werden kann. Es handelt es sich bei diesem Ansatz um eine binäre Baumstruktur an dessen Endpunkten (Blätter) sich eine Liste von UI-Komponenten (blaue Umrandung) befindet. Durch den sich davon unterscheidenden Aufbau der Desktop-UI ist es schwierig eine simple Abbildung von der einen auf die andere zu finden. Lösungen wären vermutlich komplex und zeitaufwändig, daher wurde dieser Ansatz gänzlich verworfen und nach einer Alternativen Lösung gesucht.

Umsetzung

Basierend auf der Tatsache dass eine eigene Umsetzung der Detailansicht zu aufwändig ist wird für die Darstellung die Bibliothek *react-grid-layout* genutzt. Hierfür 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 dem 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 dargestellt werden und stattdessen untereinander angezeigt werden. Außerdem sollen Komponenten eine Konfigurationsoption erhalten, mit dem Benutzer die Anzeige für bestimmte Displaygrößen aktivieren und deaktivieren können. Auch für die Übersichtsliste, welche die Form einer Tabelle hat, wird eine passende Bibliothek gesucht. Im Vergleich zur Übersichtsliste werden hier aber keine eigenen Komponenten angezeigt, sondern die Daten direkt dieser Bibliothek übergeben. Sie muss daher ein gewisses Grad an Flexibilität für die visuelle Anpassung und Formatierung bieten. Die am besten geeignetste Bibliothek *react-table* befindet sich zum Zeitpunkt der Erstellung dieser Arbeit gerade in einem großen 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 darauf diskutiert werden. Für die mobile Darstellung bietet es sich bei einer Tabelle an, alle 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 zu befüllen muss eine entsprechende Identifikation möglich sein. Es wird vorausgesetzt dass diese eindeutige ID, 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 Fehlerzustände

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 visuell ansprechend deutlich zu machen. Für diesen Zweck sollen alle *React*-Komponenten eine visuell simplere Repräsentation von sich selbst in Form von unspezifischen grauen Boxen enthalten, welche nur auf die ungefähre 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 einer Komponente



Abbildung 4.3: Mögliche Fehlerzustände einer Komponente

Mit einer entsprechenden Einfärbung und einem Hinweistext kann diese Visualisierung, wie in [Abbildung 4.3](#) an zwei unterschiedlichen Ausführungen gezeigt, ebenfalls dazu genutzt werden um Fehlerzustände beim Laden von Daten zu signalisieren.

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.

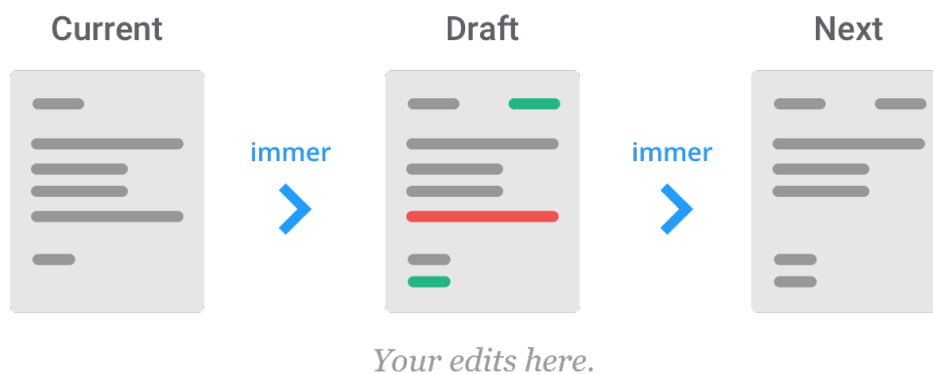


Abbildung 4.4: Editier-Konzept von immer.js [Weststrate 2019]

4.2.6. Unveränderbare Daten

Das Verwalten von Mutationen mit veränderbaren Daten ist schwieriger, je größer ein Projekt wird, da diese immer in allen Teilen des Programms ausgeführt werden müssen. Durch Programmierfehler kann es passieren, dass bestimmte Änderungen eventuell nicht überall vollzogen werden und so der State nicht mehr synchronisiert ist. Um diese Art von Fehler zu verhindern werden Daten nur an einer zentralen Stelle verwaltet (**single source of truth**) und sind an allen anderen Stellen unveränderbar (**immutable data**). Mutationen werden als Aktion an die für die Daten verantwortliche Stelle gesendet und nacheinander abgearbeitet.

4.2.7. 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.

4.2.8. Individualisierung

Die Serialisierung des Layouts nach JSON geschieht über die jeweilige Darstellungs-bibliothek der Ansicht. Zusätzliche (eigene) Formatierungen aus der Detailansicht werden anschließend im JSON ergänzt und dann 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

Die Darstellung der Übersichtsliste wird, wie oben beschrieben, die *React*-Bibliothek *react-data-grid* (siehe Abschnitt 4.2.2) benutzt. Diese ermöglicht das Suchen, Filtern und Sortieren bereits ohne weitere Anpassungen. Für die Detailansicht müssen im jetzigen Stand analog zum Desktopclient die vorhandenen Möglichkeiten im Backend zur Umsetzung der Suche und Filter genutzt werden. Ein Nachteil dieses Ansatz ist natürlich, 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 bedarf weiterer Analysen der Bibliothek.

4.2.10. Tests und Continuous Integration

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 normalen und die beschriebenen Snapshot-Tests können im Sinne der kontinuierlichen Integration¹ 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, genutzt. So ist es möglich, eine interaktive Echtzeit-Darstellung von Änderungen am Code zur direkten Beurteilung der gewünschten Auswirkungen zu nutzen.

4.3. API

FIXME: queries + mutations

4.3.1. Aufbau / GraphQL-Schema

Das Schema, welches den Aufbau der Anfragen und damit gleichzeitig der Antworten vorgibt, enthält zwei Einstiegspunkte. Zum einen können Informationen zur Übersichtsliste und zum Anderen die der Detailansicht abgefragt werden. Beide Zweige sind weiterhin in eine Struktur- und einen Datenabschnitt getrennt. Clients können so zuerst die Struktur, also alle Enthaltenen UI-Elemente und deren Platzierung, Formatierung, etc., der Ansicht abfragen. Die reine Struktur enthält im Vergleich wenige Daten, eine entsprechende Abfrage erhält also zügig eine Antwort. Während die UI anschließend anhand der Struktur aufgebaut wird und Lade-Platzhalter (siehe Abschnitt 4.2.4) anzeigt können parallel die Daten nachgeladen werden. 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.

4.3.2. Keine Business Logik im Client

Clients sollten keine eigenen Schlussfolgerungen aus den von der API gelieferten Daten ziehen müssen. Dies ist ein häufig begangener Fehler, bei dem die API Daten zu einer Ressource angibt, diese Daten aber nicht direkt angezeigt sondern erst noch auf irgendeine Art verarbeitet und transformiert werden müssen. Es kann vorkommen, dass verschiedene Implementierungen von Clients an dieser Stelle andere Transformationen

¹CI: Continuous Integration

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 beliebigen Zeitpunkt Y wieder zu verlieren. Ein Beispiel für einen solchen Fall, beschrieben von Phil Sturgeon [Sturgeon 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. Wenn nun ein weiteres Feld, welches anzeigt ob der Betrag tatsächlich auf dem Empfängerkonto eingegangen ist, **'bezahlung-erhalten-am'** hinzugefügt wird, zeigen alte Clients den Status weiterhin als 'Bezahlt', sobald das erste Feld einen Wert enthält. Clients die ein Update erhalten haben zeigen dieses 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

FIXME: move code listings to appendix

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-freundlicheres Format (JSON) übersetzt werden. Bei diesem Schritt ist es auch direkt möglich Informationen die zukünftig nicht mehr benötigt werden nicht mit zu übernehmen und andere Informationen in eine optimalere Struktur zu überführen. 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 erkennbar.

Zunächst wird für jedes einzulesende Token (XML-Element) eine Klasse vom *Acceptor-Interface* (*IDLIAcceptor* / *IVLCAcceptor*) abgeleitet, dieses Interface enthält

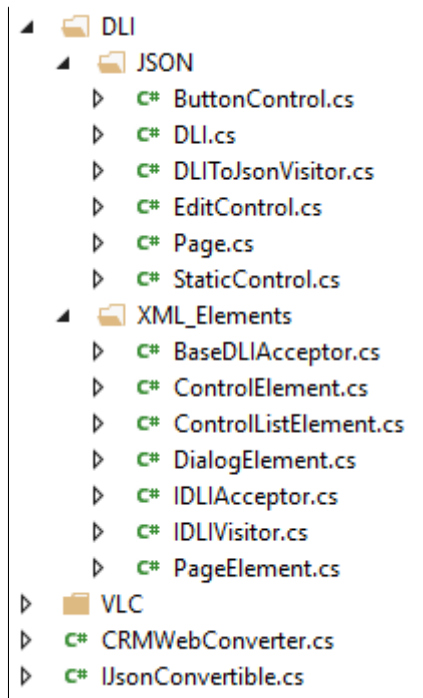


Abbildung 5.1: Datei-Struktur des Konvertierungstools

eine einzige Methode **Apply**, welche einen Visitor (*IDLIVisitor* / *IVCLVisitor*) übergeben 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
5     foreach (var childElement in element.Elements())
6     {
7         if (CheckAttribute(childElement, "name", "Title"))
8         {
9             MetaData = childElement.Value;
10            break;
11        }
12
13
14        if (CheckAttribute(childElement, "name", "Controls"))
15        {
16            ControlList = new ControlListElement(childElement);
17        }

```

```
18     }  
19 }
```

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 5.3 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

```
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  
14             completely before continuing to next page");  
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 controllList)
26 {
27     if (_currentPage == null)
28     {
29         throw new InvalidOperationException("Page must be valid to process
30             its control list");
31     }
32     foreach (var control in controllList.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
43             its controls");
44     }
45     _currentPage.AddControl(CreateControlByType(control.Type, control.
46         Items));
47 }
48 private IJsonConvertible CreateControlByType(string type, Dictionary<
49     string, string> items)
50 {
51     switch (type)
52     {
53         case "STATIC":
54             return StaticControl.ConstructWithItems(items);
55         case "BUTTON":
56             return ButtonControl.ConstructWithItems(items);
57         case "EDIT":
58             return EditControl.ConstructWithItems(items);
59         default:
60             return null;
61     }
62 }
```

Listing 5.3: Ablaufen von und Extrahieren relevanter Informationen aus XML-Tokens durch den JSON-Visitor

Die Flexibilität dieser Architektur, welche in der Abbildung 5.2 nochmals übersichtlich als Klassendiagramm dargestellt wird, ist ebenso daran zu erkennen, dass der ein-

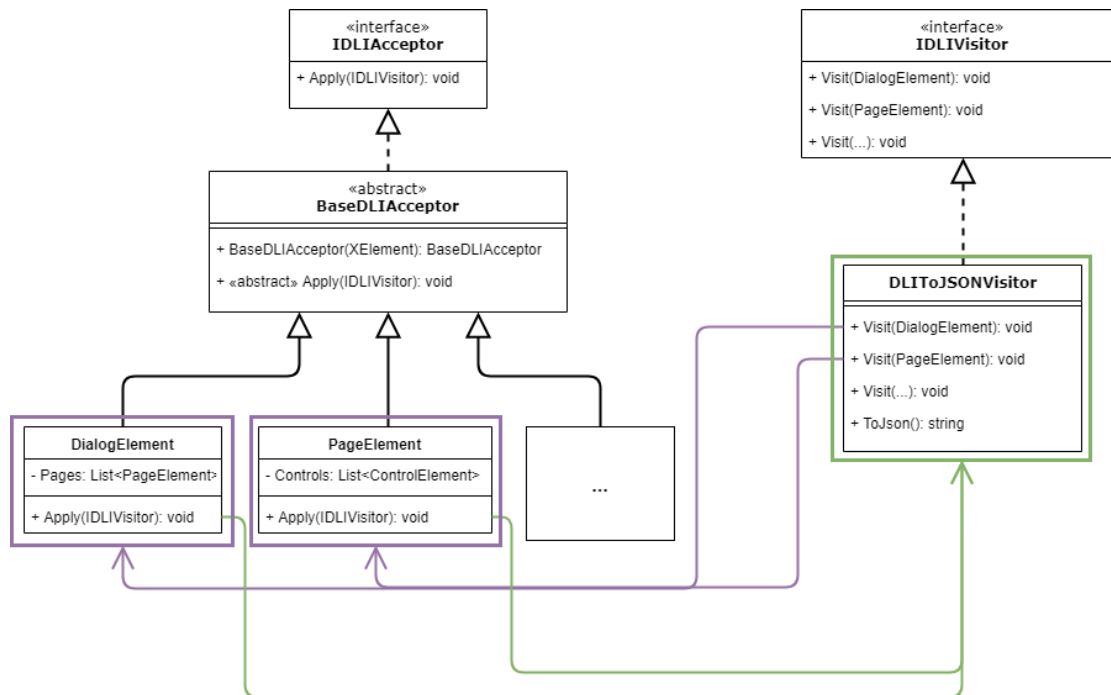


Abbildung 5.2: Klassendiagramm der Visitor-Struktur des Konvertierungstools

zige 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 ist es möglich weitere Tokens, welche eventuell zu einem späteren Zeitpunkt benötigt werden, aus den XML-Dateien auszulesen, indem weitere *Acceptor*- und *Visitor*-Implementierungen hinzugefügt werden.

Der Input und das Endergebnis in Form eines vom *GraphQL*-Server direkt verwertbaren JSON-Dokument ist in Quellcodeauszug 5.4 und Auszug 5.5 anhand eines kleinen Auszuges ersichtlich.

```

1 <list name="Page0">
2   <list name="Controls">
3     <list name="Control">
4       <item name="TypeIndex">1</item>
5       <item name="Type">STATIC</item>
6       <item name="Rect(l,t,w,h)">(18,28,420,11)</item>
7       <item name="ColorBackground">Transparent</item>
8       <item name="CondVisible">Deactivated</item>
9       <item name="InputLCID">0</item>
10      <item name="EnvironmentRestriction">0</item>
11      <item name="UIModifiers">0:UkdCKDI1NSwwLDAp,1:I1N1Z291IFVJlg==,2
          :RmFsc2U=,3:RmFsc2U=,4:RmFsc2U=,5:RmFsc2U=,6:MTMuMA==</item>
12      <item name="Font">{(255,0,0),13,-17,0,0,0,400,0,0,0,3,2,1,34,
  
```

```

        Segoe UI}</item>
13      <item name="Calculated">TRUE</item>
14      <item name="Formula"><<"--- deaktiviert seit " + Date$(
        Date(DeactivatedOn),"%02d.%02m.%04y %02H:%02i:%02s");)
        + Cond(not IsNullOrEmpty(DeactivatedReason), "; Grund: &
        quot; + DeactivatedReason) + "; ---">></item>
15      <item name="Orientation">0</item>
16    </list>
17    ...

```

Listing 5.4: XML-Input

```

1  "pages": [
2    {
3      "name": "Page0",
4      "title": "",
5      "controls": [
6        {
7          "type": "static",
8          "text": "",
9          "textFromServer": true
10         },
11         ...

```

Listing 5.5: JSON-Ergebnis

5.2. Webseite

FIXME: once project is “final” (Beispielumsetzung)

5.2.1. 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 von traditionellem CSS-Einsatz ist, dass CSS-Klassen an vielen Stellen der Oberfläche eingesetzt werden und eine Änderung am Aussehen durch den fehlenden Überblick aller Stellen möglicherweise ungewollte Auswirkungen haben kann.

Durch die Verknüpfung von CSS mit *React*-Komponenten kann sich eine Änderung immer nur auf diese eine Komponente auswirken. In Ausschnitt 5.6 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 verantwortlich, dennoch wird der jeweilige Teil der Abfrage mit den für sie relevanten Daten zusammen mit ihr definiert. Eine speziell dafür ausgelegte Komponente weiter oben in der Hierarchie kombiniert alle Teilabfragen der UI-Komponenten (siehe Ausschnitt 5.7) und sendet diese an den Server. Diese Komponente ist weiterhin dafür zuständig die vom Server erhaltenen Daten wieder auf die UI-Komponenten zu verteilen. Der vierte und letzte Bestandteil einer Komponente sind die im nächsten Abschnitt beschriebenen Tests.

```
1 // ThemedEdit can be used like a regular React component
2 const ThemedEdit = styled.input`
3   color: ${props: SimpleThemeProps} => props.theme.color}; //
4   properties depending on theme
5   width: 100%; // static css property
6   ... // additional css properties;
7 `;
```

Listing 5.6: Benutzung der styled-components Bibliothek

```
1 // query fragment of Edit component
2 const EditFragment = gql`
3   fragment EditControl on Controls {
4     ... on Edit {
5       field
6     }
7   }
8 `;
9
10 // embedding of fragments
11 const ControlsFragment = gql`
12   fragment Controls on Page {
13     controls {
14       ...EditControl
15     }
16   }
17   ${EditFragment}
18 `;
```

Listing 5.7: Einbettung von GraphQL-Query-Fragmenten

5.2.2. Tests

Einzelne Komponenten werden durch die in Abschnitt 4.2.10 erwähnten Snapshot-Vergleiche (siehe Ausschnitt 5.8) 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 nur die Konfiguration der Komponente selbst, optional werden aber jeweils noch die im Produktivcode ebenso benutzte globale CSS-Deklarationen, die verfügbaren Themes und die in der UI veränderbaren Übergabeparameter (im Ausschnitt 'Knobs'). Abbildung 5.3 zeigt die Darstellung der in Ausschnitt 5.9 gezeigten Konfiguration in der *Storybook*-Umgebung.

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

Listing 5.8: 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 5.9: Konfiguration einer Storybook Komponenten

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 (einen Auszug daraus ist in Quellcodeauszug 5.10 zu sehen) und den Resolvem die dem Schema die konkreten Daten zuordnen (der zu Auszug 5.10 passende Code kann in Auszug 5.11 eingesehen werden). Aus diesen beiden Einzelteilen wird mithilfe der *Apollo GraphQL-tools* [Apollo

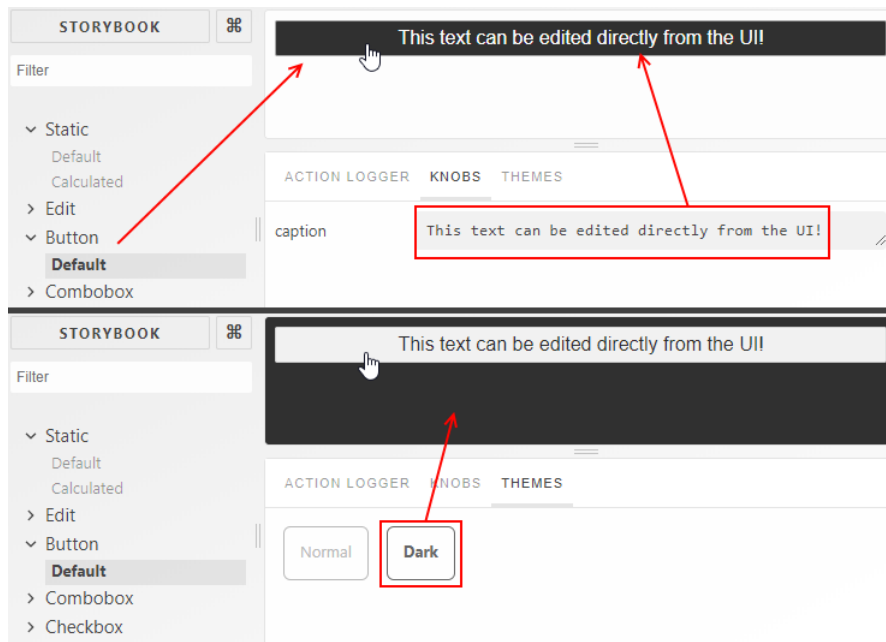


Abbildung 5.3: Beispiel für die Komponentendarstellung mit Storybook

2019] ein ausführbares Schema erstellt und mit *express-graphql* [GraphQL 2018] lokal gehostet. Für im Schema definierte Typen welche aufgrund fehlender Daten noch keine Resolver erhalten können, werden von den *GraphQL-tools* automatisch Daten mit passenden Typen generiert.

```

1 type Page {
2   title: String!
3   controls: [Controls!]
4 }
5
6 union Controls = Static | Edit | Button
7
8 type Static {
9   text: String!
10  fromServer: Boolean!
11 }

```

Listing 5.10: Teil der GraphQL-Schemadefinition

```

1 Page: {
2   title: (parent: { title: string }) => parent.title,
3   controls: (parent: { controls: {}[] }) => parent.controls
4 },
5 Controls: {
6   __resolveType(obj: { type: string }) {
7     if (obj.type === "static") {

```

```
8         return "Static";
9     }
10
11     if (obj.type === "edit") {
12         return "Edit";
13     }
14
15     if (obj.type === "button") {
16         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 5.11: GraphQL Schema-Resolver

5.3. Beispielumsetzung

Um die Validität des Konzeptes zu belegen wird eine beispielhafte, simple Oberfläche des *cRM* Desktopclients in die neue Darstellung übersetzt.

FIXME: EINFACHES (wenig Elemente), ABER KOMPLETTES BSP für jetzige UI -> neue UI

6

Fazit

Das ursprüngliche Ziel, die komplette UI umzusetzen, war zeitlich leider eine zu große Herausforderung. Die überarbeiteten Ziele, ein Konzept ausschließlich für die dynamischen Teile der *cRM* UI und den dazugehörigen Prototypen zu erstellen, wurden erreicht. Die Entwicklung mit *React* ist unkompliziert, die Auswahl an hilfreichen Bibliotheken groß und in der Community findet können zu jeder Fragestellung passende Antworten gefunden werden. *React* kann daher im weiteren Prozess ohne Bedenken weiterhin eingesetzt werden. Für ein abschließendes Fazit über *GraphQL* reicht die bisher gesammelte Erfahrung noch nicht aus, der bisherige Eindruck ist aber positiv — die Technologie sollte daher, insbesondere auch in größeren Projekten, weiter evaluiert werden. Insgesamt hat die Umsetzung des Konzepts gezeigt, dass es funktionieren kann. In Zukunft muss es weiter ergänzt werden, wobei der Prototype als gute Basis für die weitere Entwicklung dient. Persönlich hätte ich dennoch gerne mehr Zeit mit allen Teilaspekten dieser Arbeit verbracht um mehr Erfahrung mit Anforderungsformulierungen und den Technologien zu sammeln und eine fundiertere Wissensbasis für die nächsten Schritte zu haben.

6.1. Aufgetretene Schwierigkeiten

Wie oben erwähnt wurde das ursprüngliche Ziel, die komplette UI umzusetzen, nicht erreicht. Während es sich bei dieser Tatsache nicht um eine technische Schwierigkeit handelt, hatte sie dennoch weitreichende Auswirkungen auf die letztendliche Umsetzung. Auch viele der am Anfang der Arbeit antizipierten Schwierigkeiten, insbesondere

die serverbasierte Persistenz von Einstellungen und die Entscheidung über die im Web umsetzbaren Features, sind aufgrund dessen **noch** gar nicht relevant geworden.

Während der Evaluierung der Frontend-Frameworks wurde klar, dass es in diesem Bereich so viele Ressourcen zu evaluieren gibt, dass eine vollständige Analyse dieser mehr als eine eigenständige Arbeit füllen könnte. Auf der anderen Seite sind sich die Technologien in ihrer Funktionalität aber dennoch so ähnlich, dass eine Entscheidung mehr oder weniger aufgrund persönlicher und betrieblicher Präferenzen getroffen werden muss. Diesen Widerspruch korrekt darzustellen und des weiteren die korrekte Entscheidung für eine Technologie zu treffen hat mehr Zeit und Anstrengung gekostet als erwartet.

Eine weitere Schwierigkeit war es, 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. Es müssen stets alle Anforderungen im Kopf behalten werden und Lösungen gefunden werden, die allen Anforderungen zugleich gerecht werden oder zumindest keine negativen Auswirkungen auf andere Bereiche haben. Diese mentale Belastung ist mitunter sehr anstrengend und kann bei der Entscheidungsfindung lähmen, wodurch die Erreichung des Ziels erschwert ist.

6.2. Ausblick

In diesem Abschnitt soll ein Ausblick auf zukünftige Entwicklungsmöglichkeiten zum weiteren Ausbau des Projektes gegeben werden. Zunächst sollen die noch nicht fertig gestellten Funktionalitäten abgeschlossen werden. Dazu zählen zwei *React*-Komponenten, das Gruppen- und Container-Element, die Vervollständigung der Formatierungsmöglichkeiten und der Implementierung von Lade- und Fehlerzuständen aller Komponenten. Im Anschluss oder parallel kann weiter an einer optimaleren Umsetzung der *GraphQL*-API und auch der Abfragen geforscht werden. Die geringe Erfahrung in diesem Bereich lässt darauf schließen, dass hier noch Verbesserungspotential besteht.

Wenn diese Aufgaben zufriedenstellend abgeschlossen sind können die weiteren Teile der Anforderungsanalyse in Angriff genommen werden. Zum einen sind dies alle weiteren Bestandteile der UI und damit die Umsetzung als online-gehostete Webseite (inklusive der funktionalen Anforderungen an Web-Apps) und zum anderen das dafür benötigte Backend mit einem *GraphQL*-Server der mit Echtzeiten 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. ix, 33).
- 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. 12).
- Mehrabani, Afshin (o.D.). *Intro.js*. Aufgerufen am 15.03.2018. URL: <https://introjs.com/> (siehe S. 13).
- 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. 20).
- 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. 21, 22, 32).
- Weststrate, Michel (Feb. 2019). *mweststrate/immer*. Aufgerufen am 15.03.2018. URL: <https://github.com/mweststrate/immer> (siehe S. 41).
- 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. 44).
- Apollo (Jan. 2019). *apollographql/graphql-tools*. Aufgerufen am 27.01.2019. URL: <https://github.com/apollographql/graphql-tools> (siehe S. 52).

GraphQL (Okt. 2018). *graphql/express-graphql*. Aufgerufen am 27.01.2019. URL: <https://github.com/graphql/express-graphql> (siehe S. 53).

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*. O'Reilly.

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.