Lukas Greussing, BSc

Konzeption einer webbasierten Entwicklungsplattform zur Erstellung von Software-Demonstratoren, dargestellt am Beispiel der Automobilindustrie

Masterarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium: Softwareentwicklung - Wirtschaft

eingereicht an der

Technischen Universität Graz

Betreuer

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Institut für Softwaretechnologie (IST)

Mitbetreuer

Dipl.-Ing. Dr.techn. Bernhard Peischl und Dipl.-Ing. Markus Zoier

Graz, Mai 2017

Lukas Greussing, BSc

Conception of a web-based development platform for the creation of software demonstrators, illustrated by an example from the automotive industry

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Software Development and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Institute of Software Technology (IST)

Co-Supervisor

Dipl.-Ing. Dr.techn. Bernhard Peischl and Dipl.-Ing. Markus Zoier

Graz, May 2017

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Datum/Date	Unterschrift/Signature

Kurzfassung

Speziell in der Automobilindustrie stellt eine zunehmende Anzahl an interdisziplinären Entwicklungsaufgaben neue Herausforderungen an die optimale Informationsversorgung der Ingenieurinnen und Ingenieure. Ein Beispiel dafür sind hybride Antriebskonzepte (eine Kombination aus Verbrennungs- und Elektromotor), die im Zusammenspiel von Elektronik und Mechanik eine enge Zusammenarbeit verschiedener Fachbereiche erfordern. Solche Informationssysteme sind meist hochgradig komplex und werden in den einzelnen Projektphasen von zunehmend komplexeren Software-Demonstratoren begleitet. Die Schwierigkeit dabei liegt in der Bereitstellung eines möglichst kontinuierlichen Entwicklungsprozesses ohne unerwünschte Wechsel in der Technologie, damit ausgereifte Software-Demonstratoren in späten Projektphasen von einfacheren Implementierungen aus vorangegangenen Projektphasen profitieren können.

Das primäre Ziel dieser Arbeit besteht in der Erarbeitung eines Konzeptvorschlags für die Umsetzung durchgängiger Entwicklungsprozesse von Software-Demonstratoren in anwendungsnahen Forschungsprojekten der Automobilindustrie. Der theoretische Teil dieser Arbeit widmet sich zunächst, aufgrund der bewährten Einbeziehung von Expertinnen und Experten aus der Fahrzeugentwicklung, der Thematik der End-User Softwareentwicklung. Sie agieren als End-User mit Domänenwissen und wirken, speziell in den frühen Projektphasen, bei der Modellierung von Software-Demonstratoren mit. Die theoretische Analyse ausgewählter Softwarearchitekturen und geeigneter Frameworks führt anschließend zu einem Lösungskonzept, das im Wesentlichen auf zwei webbasierten Technologien aufbaut: Einerseits dem isomorphen Full-Stack JavaScript-Framework Meteor und andererseits dem clientseitigen Framework Angular. Beide Technologien werden vor diesem Hintergrund ausführlich theoretisch erörtert.

Die praktische Implementierung, welche im Zuge dieser Arbeit eigenständig entwickelt wird, soll dabei die Realisierbarkeit der Konzeptidee in Form eines Prototyps veranschaulichen und somit die theoretischen Erkenntnisse in einen konkreten, anwendungsnahen Lösungsansatz überführen. Dazu wird eine webbasierte Entwicklungsplattform mit dem Titel 'webRAD' zur Erstellung von reaktiven Software-Demonstratoren auf Basis von Meteor und Angular realisiert. Mittels webRAD wird gezeigt, dass End-User effektiv in die Entwicklung von Software-Demonstratoren integriert werden können und ein kontinuierlicher Entwicklungsprozess, ohne Technologiebrüche zwischen den einzelnen Projektphasen eines Forschungsprojekts, geschaffen werden kann.

Abstract

In the automotive industry, an increasing number of interdisciplinary development tasks brings along new challenges to the information supply for engineers. For instance, hybrid drive concepts (a combination of combustion and electric motors) require a close cooperation between the two departments of electronics and mechanics. Those usually high complex information systems are accompanied by increasingly complex software demonstrators within the individual project phases. The big challenge with it is to provide a continuous development process by avoiding changes in the used technology. Thereby advanced software demonstrators can benefit from simpler implementations from an earlier project phase.

The primary goal of this thesis is to develop a conceptual proposal of a continuous software development process in user-oriented research projects of the automotive industry. The theoretical part of this work is devoted to the topic of end-user software development, due to the proven involvement of experts from the vehicle development. They act as end-users with domain knowledge and are involved in the modelling of software demonstrators - especially in the early phases of the project. The theoretical analysis of software architectures and suitable frameworks leads to a solution concept, which essentially builds on two web-based technologies: On the one hand the isomorphic full-stack JavaScript framework Meteor and on the other hand the framework Angular. Both technologies are discussed in detail in this thesis.

The practical implementation, which has been developed in the scope of this thesis, illustrates the feasibility of the concept idea with the help of a prototype and transforms the theoretical findings into a concrete, user-oriented solution. The prototypical implementation is a web-based development platform named 'webRAD', which has been developed to create reactive software demonstrators by using Meteor and Angular. The webRAD development platform demonstrates that end-users can be effectively integrated into the development of software demonstrators and a continuous development process can be created without any technological gaps between the individual project phases of a research project.

Danksagung

An dieser Stelle möchte ich mich herzlich bei allen, die mich bei dieser Arbeit und dem vorausgegangenen Studium unterstützt haben, bedanken.

Als Erstes möchte ich Herrn Dipl.-Ing. Markus Zoier danken, der mich bereits beim Masterprojekt begleitet hat und mir den Weg zu dieser Masterarbeit in Kooperation mit dem Kompetenzzentrum - Das virtuelle Fahrzeug Forschungsgesellschaft mbH (ViF) geebnet hat. Er hat es verstanden mir die nötige Freiheit zur Gestaltung dieser Arbeit zu lassen und mich gleichzeitig mit seiner hilfsbereiten, konstruktiven Art zu motivieren und zu unterstützen.

Ein gebührender Dank gilt auch meinem wissenschaftlichen Betreuer, Herr Dipl.-Ing. Dr.techn. Bernhard Peischl, für die vielen kompetenten und fachlichen Beratungsgespräche. Von ihm habe ich wertvolle, praktische Ratschläge erhalten und genauso in vielen universitären Belangen auf seine unkomplizierte Unterstützung zählen können. Herrn Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa und dem Institut für Softwaretechnologie (IST) der Technischen Universität Graz danke ich für die Möglichkeit diese wissenschaftliche Arbeit an seinem Institut schreiben zu können.

Außerdem möchte ich mich bei Herrn Dipl.Ing. Michael Spitzer für die initiale, technische Betreuung und seine vielen hilfreichen Tipps bedanken. So mancher Lösungsansatz für ein technisches Problem war dank ihm wesentlich schneller und effizienter gefunden.

Ein besonderer Dank geht auch an meine Eltern, Großeltern und Geschwister für die Unterstützung während meines gesamten Studiums. Ein solch verständnisvoller und mitfühlender Rückhalt in der Familie war für mich sehr wertvoll. Ebenso möchte ich mich ausdrücklich bei meiner Freundin für ihre uneingeschränkte Unterstützung, ihr Verständnis und ihre Bereitschaft zum Korrekturlesen dieser Arbeit bedanken. Sie hat es immer wieder aufs Neue geschafft mich zu motivieren und zu ermutigen.

Schließlich geht mein Dank an meine Freunde, die mich bei dieser Arbeit moralisch unterstützt und mir oft Ablenkung vom Schreiballtag geboten haben. Ohne euch wäre mein Studienalltag mit Sicherheit nicht so angenehm gewesen.

Lukas Greussing
Graz, Österreich, Mai 2017

Anerkennung

Diese Arbeit entstand am Kompetenzzentrum - Das virtuelle Fahrzeug Forschungsgesellschaft mbH (ViF) in Graz, Österreich. Das ViF wird im Rahmen von COMET – Competence Centers for Excellent Technologies durch das Österreichische Bundesministerium für Verkehr und Technologie (BMVIT), das Österreichische Bundesministerium für Wissenschaft, Forschung und Wirtschaft (BMWFW), die Österreichische Forschungsförderungsgesellschaft mbH (FFG), das Land Steiermark sowie die Steirische Wirtschaftsförderung (SFG) gefördert. Das Programm COMET wird durch die FFG abgewickelt.

Außerdem möchte ich mich bei Ao.Univ.-Prof. Dr. Keith Andrews für die Erlaubnis, seine Vorlage[1] für die Erstellung dieser Arbeit verwenden zu dürfen, bedanken.

[1] Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science - http://ftp.iicm.edu/pub/keith/thesis/

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
BIRD	Builder for Illustrative and Rapid Demonstrators
CI	Continous Integration
CSP	Content Security Policy
DDP	Distributed Data Protocol
DI	Dependency-Injection
DOM	Document Object Model
DRY	Don't Repeat Yourself
ESB	Enterprise Service Bus
ESR	Extended Support Release
EUP	End-User Programmierung
EUSE	End-User Softwareentwicklung
FRP	Function Reactive Programming
IST	Institut für Softwaretechnologie
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
ORM	Object-Relation-Mapper
RPC	Remote Procedure Call
SEO	Search-Engine-Optimization
SOC	Separation of Concern
SPA	Single-Page-Applikation
UI	User Interface
ViF	Kompetenzzentrum - Das virtuelle Fahrzeug Forschungsgesellschaft mbH

WYSIWYG	What You See Is What You Get
WYSIWYT	What You See Is What You Test
XSRF	Cross-Site Request Forgery
XSS	Cross-Site Scripting
XSSI	Cross-Site Script Inclusion

Kapitel 1

Einleitung

> *“ A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work. You have to start over, beginning with a working simple system. ”*
>
> [Gall’s Gesetz, Gall, 1977, S. 52]

Die vorliegende Arbeit beschäftigt sich mit der Konzeption einer webbasierten Entwicklungsplattform zur Erstellung von Software-Demonstratoren für Forschungsprojekte in der Automobilindustrie. Speziell in der Fahrzeugentwicklung stellt eine zunehmende Anzahl an interdisziplinären Entwicklungsaufgaben neue Herausforderungen an die optimale Informationsversorgung der Fahrzeug-Ingenieure. Solche Informationssysteme sind meist hochgradig komplex und bauen daher auf Projektphasen mit zunehmend komplexeren, begleitenden Software-Demonstratoren auf. Die Schwierigkeit dabei liegt in der Bereitstellung eines möglichst kontinuierlichen Entwicklungsprozesses, damit - wie John Gall bereits 1977 treffend formulierte - komplexe Software-Demonstratoren in späten Projektphasen von einfacheren Implementierungen aus vorangegangenen Projektphasen profitieren können. Einleitend wird grob das Umfeld, die Problemstellung und die daraus resultierende Zielsetzung dieser Arbeit beschrieben.

1.1 Branche und beteiligte Partner

Die Durchführung dieser Arbeit erfolgte in enger Zusammenarbeit mit dem Kompetenzzentrum - Das virtuelle Fahrzeug Forschungsgesellschaft mbH (ViF) und dem Institut für Softwaretechnologie (IST) an der Technischen Universität Graz.

Das ViF wurde 2002 gegründet und ist ein international agierendes Forschungs- und Entwicklungszentrum, welches sich insbesondere mit der Fahrzeugentwicklung und zukünftigen Fahrzeugkonzepten für Straße und Schiene befasst. Das Kompetenzzentrum agiert als Brückenfunktion zwischen universitärer Forschung und industrieller Entwicklung und setzt sich mit Themen wie der Fahrzeugsicherheit, dem Fahrzeugkomfort bis hin zur effizienten und nachhaltigen Entwicklung und Produktion von Fahrzeugen auseinander. (Virtual-Vehicle, 2017)

Das IST der Technischen Universität Graz befasst sich sowohl in der Forschung als auch der Lehre mit theoretischen, praktischen und angewandten Aspekten der Softwaretechnologie. Der Fokus in der

Forschung liegt dabei speziell in den Bereichen intelligenter Systeme, formale Verifikation und systematisches Testen von Software, Künstliche Intelligenz, Requirements Engineering, Recommender Systeme, Optimierung von industriellen Problemen, Spieltheorie, Agile Softwareentwicklungsprozesse, Computersprachen, Computer Science Education, Management von Softwareentwicklungsprojekten (inklusive Wissensmanagement), Softwareentwicklung für Smartphones und Tablets, Kombinatorik und Komplexitätstheorie, Algorithmen und Rechnerischer Geometrie, sowie intelligente autonome Roboter. (TU Graz, 2017)

Die Relevanz von Software in der Automobilbranche hat in letzten Jahren stetig zugenommen und dieser Trend wird sich in Zukunft - insbesondere auch durch die jüngsten Fortschritte bei der Forschung und Entwicklung im Bereich des autonomen Fahrens - fortsetzen und noch verstärken. Seit den 70er Jahren wird Software zunehmend in Fahrzeugen eingesetzt. Wurden zu Beginn nur wenige Funktionen mittels Software implementiert, können aktuell die meisten Fahrzeuge als überaus komplexe und hochgradig vernetzte Systeme angesehen werden und eine Ende scheint nicht in Sicht zu sein. In naher Zukunft ist in der Automobilindustrie mit einer Zunahme der Digitalisierung und damit einhergehend mit einer steigenden Komplexität zu rechnen. Die treibenden Kräfte für diese Entwicklung liegen in der wachsenden Elektromobilität und in der Intensivierung des autonomen Fahrens. (Houdek, 2003)

1.2 Problemstellung

Eine der zentralen Herausforderungen besteht in der Entwicklung von unterstützenden Anwendungen zur optimalen, digitalen Informationsversorgung für Ingenieure aus unterschiedlichen, heterogenen Entwicklungsbereichen in der Automobilindustrie. Ein Beispiel dafür sind hybride Antriebskonzepte (eine Kombination aus Verbrennungs- und Elektromotor), die im komplexen Zusammenspiel von Elektronik und Mechanik eine enge Zusammenarbeit verschiedener Fachbereiche, sowohl in der Entwicklung als auch in der Absicherung erfordern.

Für die Entwicklung neuer IT-Konzepte in anwendungsnahen Forschungsprojekten hat sich eine agile Vorgehensweise bewährt, bei der in zeitlich kurzen Interaktionszyklen mit End-Usern die Anforderungen anhand von laufend weiterentwickelten Frontend-Demonstratoren erarbeitet werden. Beginnend mit starren Mockups werden früh erste interaktive Elemente ergänzt. Kurze Anpassungszyklen der Demonstratoren ermöglichen eine kontinuierliche Rückspiegelung neuer Teilkonzepte an die Experten in den Fachbereichen. Die Demonstratoren dienen dabei als lebendiges, erlebbares Lastenheft, welches eine Kommunikation der verschiedenen, hochspezialisierten Fahrzeug-Entwicklungsdisziplinen mit Forschern aus dem Informationsmanagement-Bereich ermöglicht. In der Folge werden die Frontend-Demonstratoren um Technologien für Datenmanagement und -integration ergänzt, sodass Piloten entstehen, die ein gesamtes Informationsmanagement-Konzept anhand repräsentativer Aufgabenstellungen zeigen.

Daraus ergeben sich typische Projektphasen innerhalb eines solchen Forschungsprojektes beim ViF, von der Anforderungserhebung, der Konzeptentwicklung über die Technologie-Evaluierung und Aufbau von Technologie-Piloten bis hin zur Implementierung von ausgereiften Software-Prototypen. Diese Projektphasen divergieren häufig in der Komplexität und in den Kommunikations- und Managementprozessen. Die wesentliche Problematik besteht nun darin, dass die begleitenden Demonstratoren gegenwärtig häufig den Einsatz verschiedener Technologien beziehungsweise Frameworks erfordern und dadurch meist technologisch getrennte Abschnitte im Entwicklungsprozess entstehen. Diese Technologiebrüche gilt es zu vermeiden, weil dadurch die Wiederverwendung von Code-Teilen von einfacheren Systemen aus vorangegangenen Projektphasen nur mit einem hohen Aufwand oder gar nicht stattfinden kann.

1.3 Zielsetzung

Die Intention dieser Arbeit besteht, ausgehend von der Analyse der Problemstellung, in der Erarbeitung eines Konzeptvorschlags für die Umsetzung kontinuierlicher Entwicklungsprozesse bei der Entwicklung von Software-Demonstratoren in Forschungsprojekten beim ViF. Ein potentieller Lösungsansatz sollte dabei die Entwicklung webbasierter, datengetriebener Anwendungen ermöglichen, welche einen verbesserten Informationszugang für komplexe, interdisziplinäre Entwicklungsaufgaben in der Automobilindustrie sicherstellen. Aufbauend auf vorhandenen Ansätzen und Technologien liegt der Fokus der Arbeit in der Analyse verschiedener Softwarearchitekturen und Technologien und zusätzlich in der prototypischen Umsetzung der erarbeiteten Konzeptidee, als Nachweis der Machtbarkeit.

Als beispielhafter Anwendungsfall an Forschungsprojekten in der Diplomarbeit dienen Datenmanagement-Lösungen für interdisziplinäre Ingenieurtätigkeiten in der Fahrzeugentwicklung, welche sich mit Informationsvernetzung beschäftigen und unter anderem folgende Anforderungen implizieren:

- Erzeugung von übergreifenden logischen Sichten
  Mittels der Bereitstellung von entsprechenden logischen Sichten, kann der Zugriff auf ein System vereinfacht werden, indem die Komplexität des darunterliegenden Systems abstrahiert wird.
- Kompatibilität mit verschiedensten Datenquellen
  Aufgrund der Informationsvernetzung von heterogenen Entwicklungsbereichen, existieren häufig unterschiedliche Datenquellen. Dabei kann es sich um relationale- oder dokumentenorientierte Datenbanken, Webdienste oder lokale Datendateien handeln.
- Rasche Erstellung von Visualisierungen für Daten aus bestehenden Systemen
  Mithilfe von aussagekräftigen Datenvisualisierungen können komplexe Zusammenhänge einfach vermittelt werden.

1.4 Aufbau der Arbeit

Im nachfolgenden Kapitel 2 wird versucht, eine Konzeptidee für die in Kapitel 1.2 geschilderte Problemstellung vorzustellen. Dazu wird zuerst das Umfeld der Arbeit genauer betrachtet, inklusive der Ausgangslage und den zentralen Anforderungen und Randbedingungen. Im Anschluss daran wird aufgrund der umfangreichen Problemstellung, welche mehrere Bereiche der Softwareentwicklung tangiert, eine möglichst gesamtheitliche Betrachtung versucht. Zunächst werden, aufgrund der bewährten Einbeziehung von Fachexperten aus der Automobilindustrie, die theoretischen Grundlagen von End-User Softwareentwicklung (EUSE) betrachtet. Diese Fachexperten agieren als End-User mit Domänenwissen und wirken, speziell in den frühen Projektphasen, bei der Modellierung von Software-Demonstratoren mit. Anschließend werden ausgewählte Softwarearchitekturen und geeignete Technologien beziehungsweise Frameworks analysiert, damit abschließend ein Lösungsansatz vorgeschlagen werden kann, auf dem in weiterer Folge die prototypische Implementierung beruht.

Die nächsten beiden Kapitel bauen auf der Konzeptidee des vorherigen Kapitels auf und widmen sich ausführlich den darin enthaltenen zentralen Technologien: Einerseits dem isomorphen Full-Stack JavaScript-Framework Meteor in Kapitel 3 und andererseits dem clientseitigen Framework Angular in Kapitel 4. Kapitel 5 befasst sich mit der prototypischen Implementierung eines webbasierten Editors zur Erstellung von reaktiven Webapplikationen basierend auf Meteor und Angular. Die praktische Implementierung, welche im Zuge dieser Arbeit eigenständig entwickelt wurde, soll dabei die Realisierbarkeit der Konzeptidee in Form eines Prototyps veranschaulichen und somit die theoretischen Erkenntnisse in einen konkreten, anwendungsnahen Lösungsansatz überführen.

Kapitel 2

Entwicklung von Software-Demonstratoren am Beispiel anwendungsnaher Forschungsprojekte in der Automobilindustrie

" Software development is technical activity conducted by human beings. "

[Wirth, 1997]

Niklaus Wirth, ein renommierter Schweizer Informatiker, beschreibt in seinem Zitat sehr treffend die Bedeutsamkeit der Ressource Mensch innerhalb der Softwareentwicklung. Die Frage nach der möglichst optimalen Einbindung unterschiedlicher Akteure in allen Phasen der Entwicklung von Software-Demonstratoren stellt ein zentraler Aspekt dieser Arbeit dar und spiegelt sich auch im implementierten Lösungsvorschlag wider.

Die Entwicklung von unterstützenden Software-Informationssystemen zur optimalen, digitalen Informationsversorgung für Ingenieure aus unterschiedlichen, heterogenen Entwicklungsbereichen in der Automobilindustrie ist eine zunehmend herausfordernde Aufgabe. In diesem Kapitel wird zuerst das Umfeld der Arbeit erläutert, ausgehend von der Ist-Situation über die zentralen Anforderungen und Randbedingungen, die speziell die Automobilindustrie bedingt. Weil die Problemstellung mehrere Bereiche der Softwareentwicklung tangiert, wird eine gesamtheitliche Betrachtung versucht und zuerst der Aspekt der End-User-Softwareentwicklung theoretisch betrachtet. Anschließend werden ausgewählte Softwarearchitekturen und mögliche Technologien beziehungsweise Frameworks genauer analysiert, damit abschließend ein in sich plausibles Konzept vorgestellt werden kann, welches bewährte Konzepte und Lösungsansätze einzubeziehen versucht.

2.1 Ausgangslage

Ein Ansatz für einen verbesserten Informationszugang sind webbasierte Cockpits, die auf einer leichtgewichtigen Aggregation zentraler Metadaten der bestehenden IT-Landschaft in den Unternehmen basieren. Derartige Ansätze sind Thema laufender und zukünftiger Forschungsprojekte, in denen das Verständnis für die Entwicklungsherausforderungen, Software-Technologien für die Lösungsfindung und die Einbeziehung des Faktors Mensch in der Produkt-Entwicklung (zum Beispiel intuitive Interaktion, kontextbasierte Aufbereitung) miteinander in Einklang zu bringen sind.

Typische Projektphasen in diesen Forschungsprojekten sind Anforderungserhebung, Konzeptentwicklung, Technologie-Evaluierung, der Aufbau von Technologie-Piloten sowie die Implementierung von

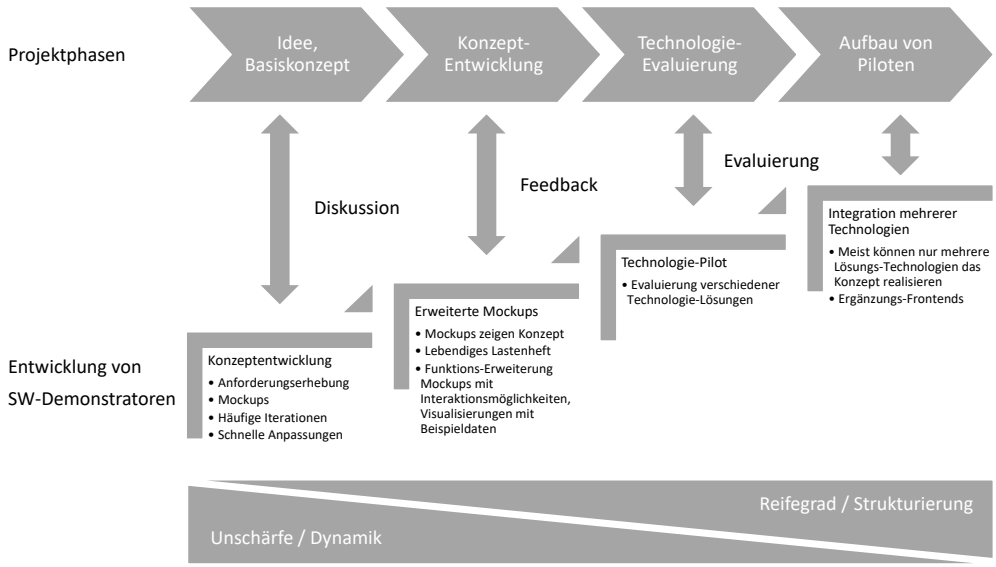

Abbildung 2.1: Typische Projektphasen in einem Forschungsprojekt der Automobilindustrie

Software-Prototypen, deren Aufbau ähnlich wie jener von Produktiv-Systemen ist. Die Lösungsfindung wird häufig durch den Einsatz von Software-Demonstratoren unterstützt, die idealerweise auf die spezifischen Bedürfnisse der Projektphasen abgestimmt sind.

Die Projektphasen, wie in Abbildung 2.1 schematisch dargestellt, divergieren in Komplexität, Technologie, Kommunikations- und Managementprozessen, die begleitenden Demonstratoren (einfache Mockups, Technologie-Piloten, Prototypen) erfordern meist den Einsatz verschiedener Demonstrator-Technologien und -Frameworks und stellen dadurch aktuell meist technologisch getrennte Abschnitte im Entwicklungsprozess dar.

In Forschungsprojekten weisen speziell die ersten Phasen eine hohe Dynamik in der Lösungsentwicklung auf, viele Iterationen in kurzen Zeitabschnitten erfordern einen sehr agilen Software-Demonstrator-Entwicklungsprozess. Hier kommen typischerweise Frontend-orientierte Mockups zum Einsatz, welche eine eng an die Abstimmungsprozesse gekoppelte, begleitende End-User-Evaluierung der Lösungsfindung ermöglichen. Bei späteren Phasen - speziell bei der Umsetzung von Technologie-Piloten und Prototypen - ist mehr Funktionalität gefordert und damit einhergehend ist eine geringere Agilität erreichbar. Dennoch sind ein möglichst nahtloser Übergang und die Wiederverwendung von Demonstrator-Bausteinen über mehrere Entwicklungsstufen wünschenswert. Im Rahmen der Einbindung vorhandener Lösungs-Technologien findet in den fortgeschrittenen Projektphasen üblicherweise eine Evaluierung vorhandener Lösungs-Technologien mittels Technologie-Piloten statt. Häufig können nur mehrere Technologien gemeinsam ein Konzept realisieren, die es abschließend im finalen Lösungs-Konzept - idealerweise basierend auf den Software-Demonstrator-Technologien - zu kombinieren und teilweise auch zu ergänzen gilt.

Tabelle 2.1 listet die zentralen Eigenschaften beziehungsweise Parameter der verschiedenen Projektphasen innerhalb eines typischen Forschungsprojektes auf. In der ersten Phase, in welcher ein Basiskonzept für eine Idee entwickelt und eine erste Anforderungserhebung stattfindet, werden üblicherweise Mockups in Form einer einfachen HTML5-Anwendung oder ganz zu Beginn sogar auf Papier erstellt. Die zentrale Anforderung besteht darin eine möglichst schnelle Bereitstellung zu ermöglichen. Daraus ergibt sich

ein sehr agiler Softwareentwicklungsprozess, der eine hohe Dynamik garantiert und die Kreativität des Teams, welches üblicherweise aus zwei Personen besteht, nicht einschränkt. Die Releasezyklen können dabei von zwei Stunden bis maximal zwei Tagen variieren. Für die Entwicklung werden Tools wie ForeUI[1] eingesetzt, mit deren Hilfe einfache, auf HTML5-basierende Webapplikationen erstellt werden können, welche jedoch nur eine statische Visualisierung von Daten ermöglichen.

Die zweite Projektphase zeichnet die zusätzliche Darstellung von exemplarischen, projektbezogenen Daten innerhalb der Webapplikation und die dafür notwendige zusätzliche Technologie in Form eines rudimentären Backends mittels Node[2] aus. Das dafür, im Rahmen meines Masterprojektes an der TU Graz in Zusammenarbeit mit dem Kompetenzzentrum - Das virtuelle Fahrzeug Forschungsgesellschaft mbH (ViF) und dem Institut für Wissenstechnologien an der Technischen Universität Graz, konzipierte Entwicklungstool Builder for Illustrative and Rapid Demonstrators (BIRD) erweitert das Mockup-Tool ForeUI mit einem dynamischen Backend, welches auf der JavaScript-basierten Server-Plattform Node aufbaut. Dazu ist clientseitig die Anpassung der Elemente in ForeUI beziehungsweise die Integration von Drittanbieter-Elementen in ForeUI notwendig gewesen, sodass eine Kommunikation zwischen der ForeUI-Anwendung und der Backend-Anwendung ermöglicht wird. Serverseitig bietet BIRD eine auf Node-basierte Backend-Anwendung, welche Daten aus unterschiedlichen Quellen (zum Beispiel CSV-Dateien oder Mockaroo[3]) beschafft, danach validiert, transformiert und mittels REST-Schnittstelle an das ForeUI-Frontend bereitstellt. Zusätzlich liefert BIRD einen Code-Generator, mithilfe dessen die Backend-Anwendung ohne Code-Kenntnisse erweitert werden kann. Das Deployment, also die Bereitstellung der mittels BIRD generierten Anwendung beim Kunden, erfolgt mittels unterschiedlicher, speziell dafür entwickelter Batchprogramme. Der begleitende Softwareentwicklungsprozess ist ähnlich wie in der ersten Projektphase durch eine hohe Dynamik und kurzen Releasezyklen von typischerweise einer Woche gekennzeichnet. Aufgrund der Abhängigkeit zu ForeUI besteht die Limitierung bei BIRD in der relativen langen Startzeit der Mockups und der Generierung von verschachteltem und komplizierten HTML5-Code, der zu einer mangelhaften Wartbarkeit und Variabilität führt.

In den letzten Projektphasen eines Forschungprojektes steht der Fokus auf der Evaluierung einer geeigneten Technologie und Implementierung von Software-Prototypen, deren Aufbau ähnlich wie jener von Produktiv-Systemen ist. Üblicherweise werden in diesen Phasen bewährte Frameworks sowohl clientseitig als auch serverseitig eingesetzt. Die Software-Prototypen verfügen über eine hohe Funktionalität und ermöglichen dadurch, eine fundierte Technologieentscheidung zu treffen. Oft werden auch Drittanwender-Technologien in das Gesamtkonzept integriert. Die Entwicklung eines Softwareprototyps erfordert einen erheblichen Entwicklungsaufwand und ist einhergehend mit einer geringeren Agilität, welches sich in Releasezyklen von typischerweise zwei Monaten widerspiegelt. Der Softwareentwicklungsprozess ist auch aufgrund der gewachsenen Teamgröße strukturierter, aber dennoch möglichst agil und flexibel. Oft werden daher in dieser Projektphase 'Scrum[4]' oder 'Kanban[5]' als Vorgehensmodell im Projektmanagement zur agilen Softwareentwicklung eingesetzt.

[1] ForeUI - Mockup-Tool - http://www.foreui.com/
[2] Node - JavaScript-basierte Server-Plattform - https://nodejs.org/
[3] Mockaroo - Datengenerator - https://www.mockaroo.com/
[4] Scrum - Vorgehensmodell im Projektmanagement - https://www.agilealliance.org/glossary/scrum/
[5] Kanban - Vorgehensmodell im Projektmanagement - https://www.agilealliance.org/glossary/kanban-board/

Projektphase	Idee Basiskonzept	Konzeptentwicklung	Technologieevaluierung / Aufbau von Piloten
Demonstrator-Typ	HTML 5 Mockup	BIRD (Builder for Illustrative and Rapid Demonstrators)	Prototyp
Releasezyklen	2 Stunden – 2 Tage typisch 1 Tag	2 Stunden – 2 Wochen typisch 1 Woche	2 Wochen - 3 Monate typisch 2 Monate
Anforderungen	Schnelle Bereitstellung, hohe Dynamik und Kreativität	Schnelle Bereitstellung, Darstellung exemplarischer Daten	Hohe Funktionalität in Frontend und Backend, ermöglicht Technologieentscheidung
Technologie	HTML5, JavaScript	HTML5, JavaScript, REST-Schnittstelle, CSV	HTML5, JavaScript, Java, Drittanwender Technologien
Tool / Framework	ForeUI	ForeUI, Node, Excel	Angular, Java
SWE Prozess	Agil, informell	Agil, eingebettet in Projektablauf	Agil, teilweise Scrum, Kanban
Teamgröße	1 - 2	1 - 3	3 - 7
Feedback	Telefonat, Email (Screenshots), Web-Konferenz	Web-Konferenz, Telefonat, Email (Screenshots)	Meeting, Web-Konferenz
Tests und Verifikation	Datenvaldidierung	-	Usability Tests
Debugging	angepasst	angepasst, Browser-Debugging	systematisch
Sicherheitsanforderungen	gering	gering	hoch
Deployment	HTML5 Generierung, Zip-Datei, Lokale Ausführung	HTML5 Generierung, Deployment-Scripts, Zip-Datei, Lokale Ausführung	Kompilierung der Module, manuelle Release-Erzeugung, VM via VPN, Download oder Paketdienst
Limitierungen	Nur statische Visualisierung von Daten	Lange Startzeit, generierter Code kompliziert und langsam	Hoher Entwicklungsaufwand, viele Schnittstellen

Tabelle 2.1: Auflistung einzelner Eigenschaften/Parameter der verschiedenen Projektphasen innerhalb eines Forschungsprojektes beim ViF

In Tabelle 2.2 werden die primären Stakeholder und deren Aufgaben und Rollen in den verschiedenen Projektphasen aufgelistet. In der ersten Projektphase sind vordringlich Experten beim ViF mit fundiertem Domänenwissen aus der Automobilindustrie und End-User involviert. Professionelle Softwareentwickler spielen in dieser Phase lediglich eine untergeordnete Rolle. Die Aufgabe des Fahrzeugexperten beim ViF besteht darin, ein einfaches Mockup, basierend auf dem Domänenwissen, zu erstellen und das Feedback der End-User hinsichtlich Benutzerfreundlichkeit zu sammeln und zu integrieren. End-User liefern dazu begleitend Ideen beziehungsweise praxisnahe Anwendungsfälle und geben Randbedingungen zum Beispiel zur IT-Infrastruktur vor.

Während der Konzeptentwicklung sind alle wichtigen Akteure involviert. Fahrzeugexperten entwickeln

das Konzept iterativ in Zusammenarbeit mit dem End-User weiter und die Aufgabe der Softwareentwickler besteht darin, spezifische Anpassungen am ForeUI-generierten Code vorzunehmen und, wenn nötig, JavaScript-Visualisierungsbibliotheken von Drittanwendern zu integrieren.

Die Rolle des Softwareentwicklers hat in der Projektphase der Technologieevaluierung und im Aufbau von Technologie-Piloten eine zentrale Bedeutung. Die Aufgaben reichen von der Recherche und Evaluierung von möglichen Technologien über die Implementierung einer funktionalen Webapplikation und häufig auch eines sogenannten Transformationslayers bis hin zur Implementierung von Schnittstellen zu Drittanwender-Technologien. Experten mit Domänenwissen übernehmen zusammen mit End-Usern in dieser Projektphase immer mehr die ebenfalls essentielle Aufgabe des Testens vom Gesamtkonzept, basierend auf praktischen Anwendungsfällen.

Projektphase / Stakeholder	Idee Basiskonzept	Konzeptentwicklung	Technologieevaluierung / Aufbau von Piloten
Software-Entwickler	-	JavaScript-Anpassungen am ForeUI-generierten Code, Einbindung von JavaScript Visualisierungsbibliotheken	Recherche und Evaluierung von Technologien, Implementierung einer Webanwendung, Implementierung eines Transformationslayers, Schnittstellen zu Drittanwender-Technologien
Fahrzeugexperte beim ViF	Modellierung eines Mockups basierend auf Domänenwissen	Iterative Konzeptentwicklung mit Kunden	Testen des Gesamtkonzeptes basierend auf Anwendungsfällen
End-User, Fahrzeugexperte beim Kunden, Ingenieur mit Domänenwissen	Liefert Ideen/Anwendungsfälle/Randbedingungen, Feedback zur Benutzerfreundlichkeit	Feedback und Weitergabe von Domänenwissen	Liefert erweiterte Anwendungsfälle, Feedback und Testen von Anwendungsfällen

Tabelle 2.2: Auflistung einzelner Stakeholder und deren Aufgaben und Rollen in den verschiedenen Projektphasen innerhalb eines Forschungsprojektes

2.2 Zentrale Anforderungen und Randbedingungen

Ein Konzept zur Erstellung eines kontinuierlichen Entwicklungsprozesses zur Unterstützung von typischen Forschungsprojekten in der Automobilindustrie ist an elementare Anforderungen und Randbedingungen gebunden.

Die Webbrowserversion wird oft vom Kunden aus Sicherheitsaspekten vorgegeben. Die meisten Kunden des ViF vertrauen dabei auf den freien Webbrowser Mozilla Firefox[6] der Mozilla Foundation in der Extended Support Release (ESR)-Version, welche über einen längeren Zeitraum mit Sicherheitsupdates versorgt wird. Zur Vermeidung von Problemen wird daher häufig eine portable Version des Webbrowsers an den Kunden mitgeliefert. Generell werden von den Kunden innovative Lösungen erwartet, welche mit webbasierten und modernen Technologien umgesetzt werden. Die Lösung muss insbesondere auch Enterprise-tauglich sein, das bedeutet sie sollte mit wenig Aufwand in die Infrastruktur einer größeren Organisation integriert werden können.

[6]Mozilla Firefox - Webbrowser - https://www.mozilla.org/

Moderne Web-Technologien entwickeln sich mit hoher Geschwindigkeit weiter. Bei der Wahl der Technologien für Software-Demonstratoren sollte dieser Aspekt daher berücksichtigt und der Fokus auf vorzugsweise zukunftssichere Technologien gelegt werden. Zusätzlich sollte auch der Aufwand für die Wartung und Aktualisierung der Software-Demonstratoren, in Anbetracht der Schnelllebigkeit der Technologien, möglichst minimiert werden.

Aufgrund der Erfahrung aus vorherigen Forschungsprojekten beim ViF mit Angular und ausgebildeter Experten in dieser Technologie ist es vernünftig, dieses Client-Framework gegenüber anderen, vergleichbaren clientseitigen Frameworks bevorzugt einzusetzen, sofern keine gravierenden Gründe dagegensprechen.

Ein Lösungsvorschlag zur Erstellung eines webbasierten Ansatzes zur Schaffung kontinuierlicher Entwicklungsprozesse muss auch berücksichtigen, sodass nicht nur in der ersten Projektphase, sondern ebenso am Ende eines Projektes stets neue Ideen und Anwendungsfälle möglich sind, welche es umzusetzen gilt. Das erfordert einen durchgehend agilen Entwicklungsprozess vom Anfang bis zum Ende eines Forschungsprojektes.

Das Deployment sollte möglichst flexibel den Bedürfnissen des Kunden angepasst werden können. Der direkte Zugriff via Internet wird oft aufgrund verschiedener Sicherheitsaspekte mit Argwohn betrachtet. Dagegen ist vielmals der Aufwand für eine sichere Webserver-Infrastruktur zu hoch oder die erforderlichen Sachkenntnisse sind nicht ausreichend gegeben. Aufgrund der Vielzahl an unterschiedlichen Szenarien beziehungsweise zur Verfügung stehenden Infrastruktur sollten möglichst alle Deployment-Varianten unterstützt werden.

2.3 Bewährte Konzepte und Lösungsansätze

Aus Erfahrungen aus früheren Forschungsprojekten ergeben sich bereits diverse bewährte Konzepte und Lösungsansätze für die Entwicklung von Software-Demonstratoren.

Zum Beispiel haben sich die strikte Trennung zwischen UI-Logik und Business-Logik und das Deployment der Webapplikation auf einem Webserver innerhalb der IT-Infrastruktur des Kunden bewährt.

Ein weiterer bewährter Aspekt besteht darin, Experten mit Domänenwissen möglichst früh in die Modellierung des Konzeptes zu integrieren und Ideen vorzugsweise vor Ort und direkt durch diese Fachexperten mittels End-User Softwareentwicklung (EUSE) umzusetzen. Im Allgemeinen sollte der gesamte Feedback-Prozess zwischen den einzelnen Stakeholdern durch eine geeignete Software unterstützt werden. Beispielsweise kann eine Funktion zum Schreiben von Kommentaren direkt in der Webapplikation eine zielgerichtete, sofortige und damit bessere Möglichkeit für Rückmeldungen ermöglichen. Im nächsten Kapitel 2.4 wird deshalb speziell das Thema EUSE genauer definiert und im Detail erläutert.

2.4 End-User-Softwareentwicklung

Das Forschungsgebiet EUSE beschäftigt sich mit der Erstellung neuer Konzepte, Technologien und Methoden, welche End-User dabei unterstützen, qualitativ hochwertige Software zu erzeugen. Die zunehmende Anzahl an Benutzern, welche selbst Programme erstellen, übersteigt bereits die Anzahl professioneller Softwareentwickler. Die daraus resultierende Software beruht zum überwiegenden Teil nicht auf bewährten Prinzipien der Softwareentwicklung und enthält oft eine Vielzahl an Fehlern. (Burnett und Myers, 2014)

Unter Softwareentwicklung werden systematische und geordnete Tätigkeiten, welche sich mit Fragen der Qualität von Software beschäftigen, verstanden. Der wesentliche Unterschied zwischen professioneller Softwareentwicklung und EUSE besteht in der Menge an Aufmerksamkeit, welche für diese qualitativen Aspekte der Softwareentwicklung aufgewendet wird. Nichtsdestotrotz müssen auch in der EUSE qualitative Aspekte der Softwareentwicklung berücksichtigt werden, diese werden jedoch meist zweitrangig behandelt. Aufgrund der unterschiedlichen Prioritäten und vereinzelt auch wegen fehlender Qualifikationen haben involvierte Personen in EUSE kaum Zeit oder Interesse an einer systematischen und geordneten Tätigkeiten der Softwareentwicklung. Die groben Unterschiede zwischen professioneller Softwareentwicklung und EUSE werden in Tabelle 2.3 zusammengefasst. EUSE kann demnach vereinfacht durch seine ungeplante, implizite und opportunistische Art charakterisiert werden. (Ko u. a., 2011)

Tätigkeiten in der Softwareentwicklung	Professionelle Softwareentwicklung	End-User Softwareentwicklung
Anforderungen	explizit	implizit
Spezifikationen	explizit	implizit
Wiederverwendbarkeit	geplant	ungeplant
Tests und Verifikation	vorsichtig	ungeplant
Debugging	systematisch	opportunistisch

Tabelle 2.3: Qualitative Unterschiede zwischen professioneller Softwareentwicklung und End-User-Softwareentwicklung [Tabelle wurde adaptiert von Ko u. a. (2011)]

Aufgrund dieser Unterschiede besteht die Herausforderung darin, Wege und Möglichkeiten zu finden, systematische Softwareentwicklungstätigkeiten in den bestehenden Arbeitsablauf der Benutzer zu integrieren, ohne die Art der Arbeit oder die Prioritäten wesentlich zu beeinflussen. Zum Beispiel statt zu erwarten, dass ein Benutzer einer Tabellenkalkulation selber Tests integriert, könnte ein Tool die erfolgreichen und fehlgeschlagenen Eingaben schrittweise verfolgen und Rückmeldungen hinsichtlich der Qualität der Software bereitstellen. Solche Lösungsansätze erlauben es, dass Benutzer sich auf die primären Ziele der Software konzentrieren können und gleichzeitig qualitativ hochwertige Software erstellen können. (Ko u. a., 2011)

2.4.1 Einbettung von Softwareentwicklungstechniken

Eine der größten Herausforderungen von EUSE besteht darin, End-User zu motivieren, nicht vertraute Techniken der Softwareentwicklung über den gesamten Lebenszyklus einer Anwendung hinweg einzusetzen. Klassische Aktivitäten in der Softwareentwicklung umfassen die Bereiche: Anforderungsanalyse, Design und Spezifikation, Testen und Verifikation, Debugging und Wartung. Ebenso stellt der Aspekt der Wiederverwendung von bereits existierendem Code ein wesentliches Prinzip in der Softwareentwicklung dar. In den folgenden Paragraphen wird auf die Probleme für End-User und auf bereits existierende Lösungsansätze in den jeweiligen Bereichen der Softwareentwicklung eingegangen. (Lizcano u. a., 2013)

Zu Beginn des Lebenszyklus bei der Softwareentwicklung beschreiben End-User häufig nicht, wie sich die spätere Software verhalten soll. Die Anforderungsanalyse wird meist als unnötiger Aufwand betrachtet, was in weiterer Folge zu erheblichen Problemen in späteren Phasen des Entwicklungsprozesses führen kann. Ein existierender Lösungsansatz besteht darin, dass spezielle Fragen oder ein ähnlicher Mechanismus angewendet wird, mithilfe dessen die gewünschten Anforderungen entlockt werden können. (Lizcano u. a., 2013)

Der Schritt von einer Liste an Anforderungen zu einem Design-Konzept und einer Spezifikation, welche die Anforderungen abdeckt, ist oft der schwierigste Teil der Softwareentwicklung und ist umso kompli-

zierter in Verbindung mit EUSE. Möglichst genau zu spezifizieren, wie die Software intern aufgebaut wird, damit sie alle Anforderungen erfüllt, ist für End-User ungleich schwieriger, weil sie oft nicht über die notwendigen Techniken und Heuristiken verfügen. Für dieses Problem in der EUSE wurden in der Literatur bereits mehrere Lösungen vorgeschlagen. Zum Beispiel kann eine für End-User verständliche Spezifikationssprache, ein sogenannter What You See Is What You Get (WYSIWYG)-Editor oder ein vordefinierter Satz an Design- beziehungsweise Spezifikationskomponenten bereitgestellt werden. Letzten Endes werden oftmals auch externe Experten für diesen Bereich der Softwareentwicklung miteinbezogen. (Lizcano u. a., 2013)

Der Aspekt der Wiederverwendung von bereits existierendem Code, um damit Zeit zu sparen und Fehler zu vermeiden, stellt sich häufig für End-User als mühevoller dar, als eine Anwendung von Grund auf neu zu implementieren. Dabei führt oft genau die Wiederverwendung von bestehen Implementierungen oder Konzepten zu der gewünschten Lösung. Ein vorgeschlagener Ansatz für das Problem ist die Verwendung von Katalogen aus Lösungen beziehungsweise Komponenten, welche mittels Such- und Empfehlungsmechanismen unterstützt werden, damit End-User diese Lösungen besser verstehen und sie später in ihrer spezifischen Domäne effektiv wiederverwenden können. (Lizcano u. a., 2013)

Bei der Implementation besteht die größte Problematik darin, dass End-User ohne die notwendigen Programmierkenntnisse diese Aufgabe schlicht nicht ausführen können. Abhilfe können hier die intensive Wiederverwendung von Software und/oder der Einsatz von Design-Aktivitäten bei der Implementierung geben. (Lizcano u. a., 2013)

Das Testen von Software erhöht das Vertrauen in die Korrektheit der Software und der Identifizierung von Fehlern. End-Usern fehlt jedoch häufig das Wissen und die Erfahrung Tests zu erstellen und auszuführen. Ein übermäßiges Vertrauen in die Korrektheit der selbst erstellten Lösungen ist oft der Hauptgrund für schlechte Qualität im Bereich der EUSE. In der Literatur wird dazu die Verwendung von systematischen Test-Tools oder eine What You See Is What You Test (WYSIWYT) White-Box Methodik vorgeschlagen. White-Box-Tests bezeichnen Tests, welche mit Kenntnissen über die innere Funktionsweise des Systems, erstellt werden. Ein weiterer Ansatz besteht darin, dass die Lösungs-Spezifikation automatisch mit dem generierten Ergebnis der End-User, mittels speziellen Prüfsystemen, verglichen wird. (Lizcano u. a., 2013)

Die automatische Generierung von Testfällen für den Endkunden kann ebenfalls die Qualität der Anwendung bedeutend erhöhen. Musliu u. a. (2013) untersuchten diesbezüglich die automatische Testgenerierung für Programme, welche in Visual Basic geschrieben und in Microsoft Excel verwendet werden. Die Autoren schlagen dabei eine Methodik vor, welche auf einem heuristischen Suchverfahren für die automatische Generierung von Testfällen basiert. Das Verfahren stellt zudem sicher, dass eine hohe Testabdeckung erreicht wird, indem der Code direkt in Microsoft Excel in verschiedenen Farben visualisiert wird, sodass für End-User erkennbar ist, welcher Teil des Codes bereits mit Testfällen abgedeckt wurde. (Musliu u. a., 2013)

Mittels Debugging können bekannte Fehler lokalisiert und behoben werden. End-User ignorieren häufig diese Möglichkeit und konzentrieren sich darauf, so schnell wie möglich eine lauffähige Anwendung zu erstellen. Ein möglicher Ansatz könnte die Verwendung von interaktiven Assistenten sein, welche die Kontroll- und Datenabhängigkeiten verwalten und damit End-User unterstützen. (Lizcano u. a., 2013)

Nachdem in diesem Kapitel grob auf die Probleme für End-User in den verschiedenen Bereichen der Softwareentwicklung eingegangen wurde und dazu existierende Lösungsansätze aus der Literatur für die Einbettung von Softwareentwicklungstechniken im Forschungsgebiet EUSE aufgezeigt wurden, wird im nächsten Kapitel das verwandte Forschungsgebiet End-User Programmierung (EUP) kurz erläutert und versucht, die Unterschiede hervorzuheben.

2.4.2 Abgrenzung zu End-User-Programmierung

Der Ausdruck EUP wurde durch Nardi (1993) in einer Studie über die Verwendung von Tabellenkalkulationen bei der Arbeit populär. Vereinfacht dargestellt, kann jeder Benutzer eines Computers als End-User betrachtet werden. EUP ist demnach dadurch charakterisiert, dass das Resultat der Programmierung maßgeblich einen persönlichen und keinen öffentlichen Zweck hat. Die essentielle Unterscheidung besteht im Wesentlichen darin, dass ein solches Programm üblicherweise nicht für eine größere Anzahl an Personen mit unterschiedlichen Anforderungen erstellt wird. Diese Definition trifft gleichermaßen auch auf professionelle Softwareentwickler zu, die zum Beispiel ein Hilfstool zur Unterstützung einer ganz speziellen Aufgabe implementieren, welche nicht für eine erweiterte Gruppe an Benutzern oder Anwendungsfällen bestimmt ist. (Ko u. a., 2011)

Im Gegensatz zu EUP ist das primäre Ziel professioneller Softwareentwickler Anwendungen für andere Benutzer zu entwickeln. Die Intention hat meist ökonomische Gründe. Es können jedoch auch andere Absichten, wie sie dies beispielsweise bei Open-Source-Projekten der Fall ist, dahinter stehen. Demzufolge ändert sich die Definition der Aktivität, sobald ein begeisterter Webdesigner sich entscheidet, nicht nur eigene Webseiten zu gestalten, sondern diese Tätigkeit auch für andere Interessierte durchführt oder ein Entwickler zum Beispiel ein Visualisierungstool für Datenstrukturen nicht nur für sich, sondern auch dem Rest seines Teams oder der gesamten Organisation zur Verfügung stellt. In diesem Moment ändert sich die dahinterliegende Absicht und Entwickler müssen die Anwendung für einen größeren Rahmen an möglichen Anwendungsfällen planen und entwerfen. Gleichzeitig nimmt die Bedeutung der Softwarearchitektur, dem Testen der Software und die Anzahl an möglichen Fehlerquellen erheblich zu. (Ko u. a., 2011)

Diese im vorherigen Absatz erläuterte Definition von EUP kann nicht immer isoliert betrachtet werden. Schließlich ist oft keine präzise Abgrenzung zwischen einem Programm für drei Benutzern und einem Programm für 100 Benutzern gegeben. Stattdessen liegt die Differenzierung darin, dass je mehr Benutzer für das jeweilige Programm vorgesehen sind, desto mehr Aspekte aus der professionellen Softwareentwicklung berücksichtigt werden müssen, damit die zunehmend komplexeren Anforderungen erfüllt werden können. Vereinzelt ändern sich auch die Umstände einer Anwendung. Ein Programm, welches zuerst nur für einen Zweck programmiert wurde, wird ausgeweitet und für viele Anwender zugänglich gemacht. Der Code, welcher womöglich zuvor ungetestet und unüberlegt implementiert wurde, erfordert dann eine strengere Betrachtung hinsichtlich bewährter, professioneller Softwareentwicklungspraktiken. (Ko u. a., 2011)

Für den Begriff EUP existieren in der Literatur mehrere Definitionen. Viele setzen den Begriff in Verbindung mit 'Programmierung für Anfänger' beziehungsweise 'nicht professionelle Programmierung' oder beschreiben damit die Identität eines Individuums. Diese Definitionen setzten sich laut Ko u. a. (2011) aus verwandten aber nicht gleichwertigen Konzepten zusammen. Zum Beispiel kann ein Buchhalter bei sich Zuhause eine Tabellenkalkulation für den Überblick seiner Haushaltsfinanzen erstellen. In der Arbeit wird er jedoch eine Tabellenkalkulation verwenden, welche mit anderen Buchhaltern geteilt werden kann und die gemeinsame Verwaltung von steuerlichen Aktivitäten ermöglicht. Es wäre demnach unzutreffend diesen Buchhalter in allen Situation als End-User zu bezeichnen, weil die Tabellenkalkulation für die Haushaltsfinanzen lediglich einen persönlichen, kurzfristigen Zweck verfolgt und die Tabellenkalkulation bei der Arbeit intensiv gewartet und von mehreren Personen verwendet wird. Genauso entscheidend ist es den Begriff EUP nicht mit dem Begriff 'Unerfahrenheit' gleichzusetzen. Professionelle Softwareentwickler mit jahrelanger Erfahrung können aktiv EUP betreiben, indem sie Programme für persönliche Zwecke erstellen - ganz ohne Intention sie mit anderen zu teilen. (Ko u. a., 2011)

Bei vielen Softwarebenutzer sind die Anforderungen, welche sie an die Software stellen, unterschiedlich komplex und ändern sich in regelmäßigen Abständen. Professionelle Softwareentwickler können oft

diese Erfordernisse nicht ausreichend abdecken, weil sie nicht über das nötige Fachwissen innerhalb der Berufsgruppe verfügen und der Entwicklungsprozess zu lange dauert. EUP hilft das Problem zu lösen, indem End-User in die Lage versetzt werden, selber ein Programm zu erstellen, welches deren Anforderungen abdeckt. (Scaffidi u. a., 2012)

2.5 Ausgewählte Softwarearchitekturen

In diesem Kapitel werden einige ausgewählte Softwarearchitekturen, welche häufig in webbasierten Anwendungen eingesetzt werden, theoretisch betrachtet. Dabei wird kein Anspruch auf Vollständigkeit erhoben, sondern lediglich ein grober Überblick vermittelt. Im Wesentlichen wird zwischen der Service-orientierten, modernen Microservice-Softwarearchitektur in Kapitel 2.5.1 und den eher monolithischen und bewährten Model-View-Architekturmustern in Kapitel 2.5.2 unterschieden. Abschließend wird auf die Vor- und Nachteile eingegangen und versucht eine geeignete Architektur für die Entwicklung von Software-Demonstratoren in Forschungsprojekten des ViF vorzuschlagen.

2.5.1 Microservice-Architektur

Microservices lassen sich grob charakterisieren als kleine, autonome Services, welche in einem eigenen Prozess laufen und durch einfache, standardisierte Mechanismen - häufig HTTP - miteinander kommunizieren. Diese Services orientieren sich entlang von Geschäftseinheiten und können jeweils unabhängig voneinander und jederzeit bereitgestellt werden. Die einzelnen Microservices können unterschiedliche Programmiersprachen und Datenbanken verwenden und unterliegen nur einem Mindestmaß an zentraler Verwaltung. (Fowler und Lewis, 2015)

Abbildung 2.2 illustriert die Unterschiede und das Verhalten bei der Skalierung zwischen monolithischen Anwendungen und der Microservice-Architektur. Monolithische Systeme werden häufig in drei unterschiedlichen Ebenen realisiert: Der Benutzeroberfläche, der Datenbank und einer serverseitigen Anwendung. Diese serverseitige Anwendung wird als Monolith bezeichnet, also eine logisch einzeln ausführbare Datei, in welcher alle Funktionen in einem Prozess laufen. Eine Skalierung ist meist nur durch die Replikation des gesamten Monolithen möglich. Im Gegensatz dazu können Microservices aufgrund ihrer Autonomie und der losen Kopplung, wenn nötig einzeln auf verschiedene Server repliziert werden. (Fowler und Lewis, 2015)

Der Wunsch Softwaresysteme mittels Komponenten zusammenzusetzen, ähnlich wie in der physischen Welt, besteht in der Softwareentwicklung schon sehr lange. Der Begriff der Komponente zu definieren ist dabei gar nicht so trivial. Die am meiste zitierte Definition lautet: Eine Komponente ist eine Softwareeinheit, welche autonom austauschbar und erweiterbar ist. Innerhalb von Microservices können auch Bibliotheken eingesetzt werden, welche ebenfalls als Komponenten betrachtet werden können. Die primäre Komponententrennung in der Microservice-Architektur erfolgt jedoch mittels Aufsplittung in einzelne Services. Der wesentliche Vorteil dabei besteht im Unterschied zu Bibliotheken, dass Services unabhängig bereitgestellt werden können. Zum Beispiel müsste eine Anwendung mit mehreren Bibliotheken, welche alle innerhalb eines Prozesses laufen, nach einer Änderung in einer einzelnen Komponente die gesamte Anwendung neu bereitstellen. Im Gegensatz dazu genügt bei einem System, welches aus Microservices besteht, die Bereitstellung des entsprechenden Service, in welchem Änderungen vorgenommen wurden. Der Nachteil die Komponenten mittels Services zu realisieren besteht darin, dass verteilte Aufrufe teurer sind als Aufrufe innerhalb eines Prozesses. Deshalb sind APIs bei Microservices häufig sehr allgemein definiert, was deren Benutzung umständlicher macht. (Fowler und Lewis, 2015)

Typischerweise wird die Entwicklung einer Applikation in Teams für die Benutzeroberfläche, der Backend-Logik und der Datenbank aufgeteilt. Diese Tatsache führt aufgrund des nach Conway formulierten Gesetz zu einer Applikation, welche ebenfalls diese Struktur abbildet:

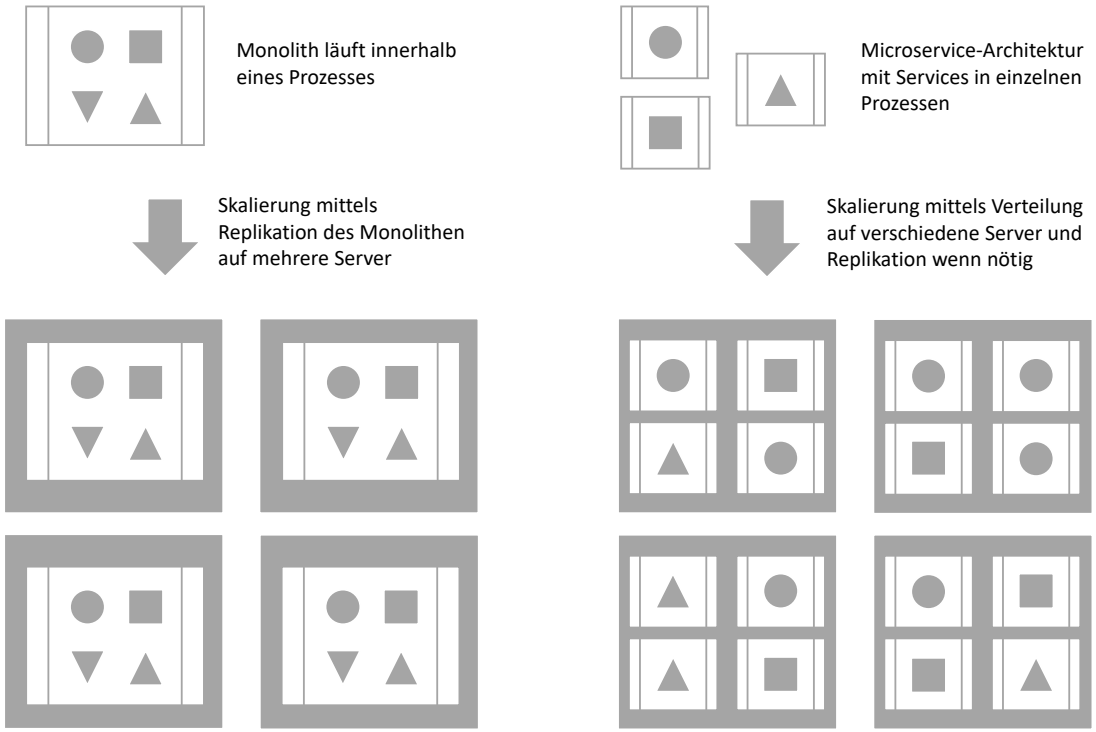

Abbildung 2.2: Vergleich des Skalierungmodells von Monolithen und Microservices [Abbildung adaptiert von Fowler und Lewis (2015)]

> " Organisationen, die Systeme entwerfen (im weitesten Sinne), sind auf Entwürfe festgelegt, welche die Kommunikationsstrukturen dieser Organisation abbilden. " (Conway, 1968)

Bereits kleine Änderungen in einer derart strukturierten Anwendung bedarf häufig die Einbindung von Entwicklern über Teamgrenzen hinweg, was einen erhöhten Zeit- und Kostenaufwand bedeutet. Die Microservice-Architektur verfolgt diesbezüglich einen anderen Ansatz und versucht, das System nach Geschäftseinheiten beziehungsweise Geschäftsressourcen zu strukturieren. Dabei implementieren solche Services Funktionalitäten für die entsprechende, fachlich getrennte Einheit über den gesamten Technologie-Stack, also inklusive der Benutzeroberfläche, der Datenspeicherung und der Kommunikation mit anderen Services des Systems. Dadurch arbeiten die zuständigen Teams funktionsübergreifend und verfügen über Kompetenzen in allen Bereichen der Softwareentwicklung. (Fowler und Lewis, 2015)

Ein Ziel von Microservices besteht darin, möglichst autonom und in sich kohäsiv zu sein. Es wird daher der Ansatz von intelligenten Endpunkten und dummen Verbindungen (Smart Enpoints and dump Pipes) verfolgt. Ein Beispiel für eine gegensätzliche Vorgehensweise ist der Enterprise Service Bus (ESB). Solche Systeme weisen meist umfangreiche Kommunikationsmechanismen mit viel Intelligenz, wie zum Beispiel Message-Routing, Transformation und die Implementierung von spezifischer Geschäftslogik auf. Microservices beinhalten dagegen ihre eigene Geschäftslogik und agieren ähnlich wie Filter im klassischen Sinne - nach empfangen einer Anfrage, wird diese bearbeitet und eine Antwort versendet. In der Architektur der Microservices werden daher oft Protokolle und Prinzipien des Internets wie zum Beispiel das HTTP-Protokoll und ein leichtgewichtiges Messaging wie zum Beispiel RabbitMQ oder ZeroMQ verwendet. (Fowler und Lewis, 2015)

Aufgrund der Strukturierung nach Geschäftseinheiten und dem Ansatz von intelligenten Endpunkten und dummen Verbindungen von Microservices wird der Einsatz von unterschiedlichen Technologien in den jeweiligen Services ermöglicht. Dadurch können die für den entsprechenden Zweck am besten ge-

eignetsten Tools und Technologien eingesetzt werden, anstelle von standardisierten Lösungen, welche häufig einen Kompromiss darstellen. Zum Beispiel könnte in einem Service, aufgrund von Performance-Anforderungen, ein komplett anderer Technologie-Stack angewendet werden, der eine optimale Lösung für das Problem erlaubt. Zusätzlich ermöglichen Microservices neue Technologien schneller und einfacher zu adaptieren. Natürlich birgt eine unlimitierte Anzahl an unterschiedlichen Technologien auch Gefahren und Kosten, weshalb viele Unternehmen Beschränkungen zum Beispiel hinsichtlich der Wahl von Programmiersprachen vorgeben. (Newman, 2015)

Ein weiterer Vorteil von Microservices ist die zusätzliche Ausfallsicherheit des gesamten Systems. Wenn eine Komponente fehlschlägt sorgen Grenzen zwischen Services dafür, dass der Fehler isoliert wird und der Rest des Systems weiterarbeiten kann. In einem monolithischen System können Ausfälle reduziert werden, indem das gesamte System auf mehreren Servern redundant ausgeführt wird, wohingegen mit Microservices das System den Fehler behandelt und die betroffene Funktionalität entsprechend aussetzt. Andererseits weisen verteilte Systeme neue Fehlerquellen auf, welche es zu beachten gilt. Netzwerke können genauso wie Computer ausfallen oder Verzögerungen verursachen. Verteilte Systeme müssen daher eine gewisse Fehlertoleranz aufweisen, damit sie auf solche Fehler entsprechend reagieren können. Ein möglicher Ansatz wäre zum Beispiel das 'Fail Fast'-Prinzip, also möglichst früh auf bekannte Fehler zu reagieren, oder die Verwendung von geeigneten Timeouts, welche stetig überprüft und aufgezeichnet werden. (Newman, 2015)

Bei Microservices handelt es sich um verteilte Systeme, die aufgrund ihrer Natur eine zusätzliche Komplexität implizieren. Jedes Unternehmen, jede Organisation und jedes System ist unterschiedlich und viele Faktoren spielen für die Wahl der optimalen Softwarearchitektur und den jeweiligen Zweck eine Rolle. Bei Microservices handelt es sich um keine Wunderlösung für jede Situation. Im nächsten Kapitel 2.5.2 werden deshalb auch klassische, monolithische Softwarearchitekturen für webbasierte Anwendungen genauer betrachtet. (Newman, 2015)

2.5.2 Model-View-Architekturmuster

Jede Software, welche mit Benutzern interagiert, benötigt irgendeine Art von Benutzeroberfläche. Model-View-Controller (MVC) ist ein weithin bekanntes, strukturelles Entwurfsmuster, welches dabei hilft die Benutzeroberfläche mit der Geschäftslogik zu verknüpfen. Das Entwurfsmuster separiert die Repräsentation der Geschäftslogik (Model) von der Anzeige der Anwendung (View) und der Verwaltung von Benutzerinteraktionen (Controller). In der Literatur existieren eine Vielzahl an Entwurfsmuster, wie sie in Abbildung 2.3 schematisch dargestellt sind, welche von der MVC-Architektur abgeleitet werden können. Alle Entwurfsmuster basieren auf dem Separation of Concern (SOC)-Prinzip, also der strikten Trennung von Aufgabenbereichen. Das Beachten dieses grundlegenden Prinzips in der Softwareentwicklung führt zu einem klaren Design, welches einfacher zu warten und zu testen ist. Die drei wichtigsten Familien dieser Softwarearchitekturen MVC, Model-View-ViewModel (MVVM) und Model-View-Presenter (MVP) werden in diesem Kapitel kurz erläutert. (Syromiatnikov und Weyns, 2014)

Model-View-Controller (MVC)

Bei MVC handelt es sich um das einflussreichste Entwurfsmuster für die Synchronisation der Benutzeroberfläche mit der Geschäftslogik einer Anwendung. Es wurde bereits 1980 von Krasner und Pope formuliert. Anfänglich wurde MVC für die Implementierung von Desktop-Applikationen mit einer umfangreichen, grafischen Benutzeroberfläche verwendet. Im Laufe der Zeit entwickelte sich das ursprüngliche MVC-Entwurfsmuster weiter und es entstanden, aufgrund von neuen Technologien, welche neue Anforderungen implizierten, Varianten des originalen Entwurfsmusters. Heute wird MVC in den verschiedensten Bereichen eingesetzt, wie zum Beispiel in Webanwendungen oder mobilen Applikationen.

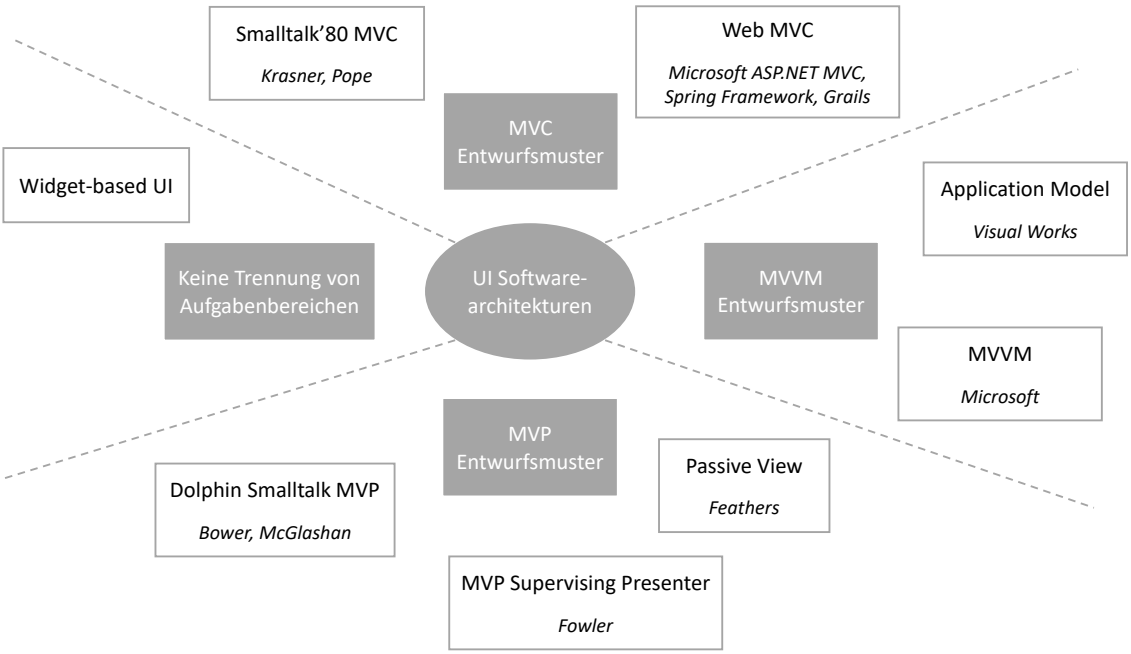

Abbildung 2.3: Überblick über die Familie der Model-View-Architekturmuster [Abbildung adaptiert von Syromiatnikov und Weyns (2014)]

Speziell in der Entwicklung von Webanwendungen muss zwischen serverseitigem MVC und clientseitigem MVC unterschieden werden. Ursprünglich wurde das Entwurfsmuster vorwiegend auf dem Server eingesetzt, so beispielsweise in den Frameworks ASP.NET MVC, Spring oder Grails. Inzwischen existieren jedoch auch Frameworks für den Client - wie zum Beispiel Backbone, Knockout oder Angular. Bei diesen Framewoks handelt es sich jedoch häufig nicht um herkömmliche MVCs, weshalb in diesem Kapitel klassische serverseitige MVC-Muster betrachtet werden. (Syromiatnikov und Weyns, 2014)

Die Intention von MVC liegt in der Separation der Geschäfts- von der Präsentationslogik mithilfe von drei Komponenten mit jeweils eindeutigen Verantwortlichkeiten, wie es in der Abbildung 2.4 gut erkennbar wird: Die View-Komponente für die Generierung des HTML-Layouts, die Controller-Komponente zur Verwaltung der Benutzerinteraktionen und die Model-Komponente zur Speicherung von Daten. Klassische Webapplikationen unterstützen von Natur aus die Separation zwischen der View und dem Controller, indem Daten clientseitig als HTML dargestellt werden und die bereitgestellten Daten serverseitig verarbeitet werden. Die Anwendungslogik wird vom Controller angestoßen, während das Model lediglich die Daten, welche in der View benötigt werden, speichert. Bei einem klassischen MVC für Desktopanwendungen wird ein sogenannter 'observer' für die direkte Synchronisation zwischen der Benutzeroberfläche und den Daten verwendet. Klassische Webapplikationen operieren zustandslos und eine enge Synchronisation ist daher nicht erforderlich. Erkennbar wird dieser Aspekt, wenn ein typischer Prozessablauf betrachtet wird. Zum Beispiel, wenn ein Benutzer eine Webseite mit Artikeln besucht und zusätzliche Informationen dazu erhalten möchte. Durch Eingabe der Adresse im Browser wird eine Anfrage an den Server gesendet. Der Server aktiviert daraufhin den zu dieser Adresse gehörenden Controller, welcher anschließend entsprechende Operationen ausführt, damit die gewünschten Artikel zurückgeliefert werden können. Danach erstellt der Controller üblicherweise ein Model und initialisiert es mit der Liste an Artikeln. In der Folge wird das Model an die View gesendet und dort dem Benutzer dargestellt. (Syromiatnikov und Weyns, 2014)

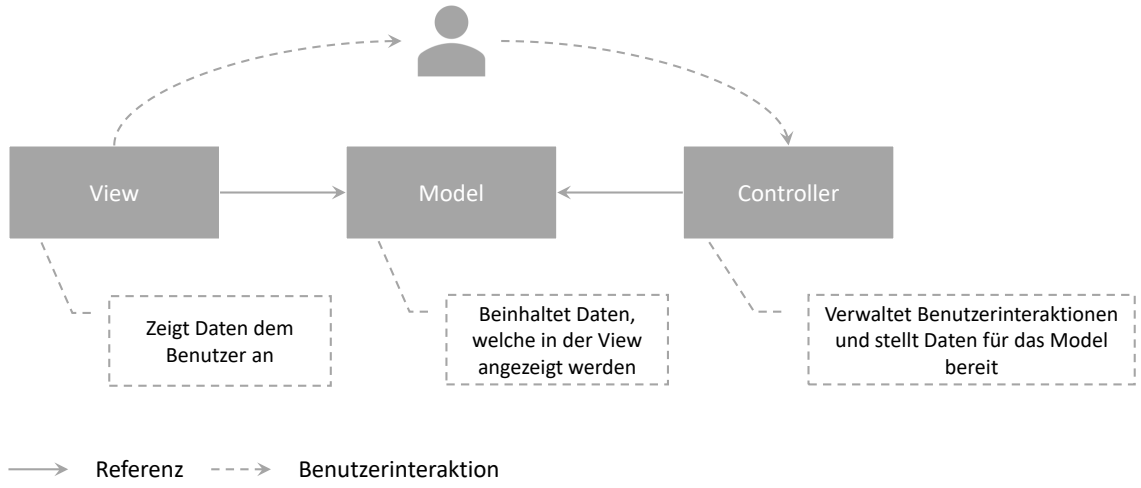

Abbildung 2.4: Web MVC Entwurfsmuster [Abbildung adaptiert von Syromiatnikov und Weyns (2014)]

Model-View-ViewModel (MVVM)

Das MVVM-Entwurfmuster ist eine Abwandlung des MVC-Entwurfsmusters. Microsoft machte es populär und entwickelte Technologien, wie WPF oder Silverlight, welche eng mit MVVM verknüpft sind. Die Intention von MVVM liegt ebenso wie bei MVC in der Separation der Geschäftslogik von der Präsentationslogik. Abbildung 2.5 zeigt die drei Komponenten von MVVM mit jeweils klar definierten Aufgaben. Die Model-Komponente dient der Verwaltung der Geschäftsdaten, die View-Model-Komponente dient der Verwaltung des View-Zustandes und der Benutzerinteraktionen und die View-Komponente ist für die Darstellung der Benutzeroberfläche zuständig. Ein wesentlicher Unterschied zu MVC besteht darin, dass die View- und die View-Model-Komponente deklarativ, mittels der Observer-Synchronisation, miteinander verbunden sind. (Syromiatnikov und Weyns, 2014)

Bei MVVM kann eine View-Model-Komponente über mehrere Views verfügen. Dadurch können gleiche Daten simultan auf verschiedene Arten dargestellt werden. Das Entwurfsmuster hat eine lineare Struktur. Die View-Komponente ist für die Darstellung der Benutzeroberfläche verantwortlich. Dabei kann sie die View-Model-Komponente observieren, indem sie deren Methoden aufruft oder falls nötig deren Eigenschaften ändert. Für diesen Zweck verfügt die View über eine Referenz in eine Richtung: Von der View zum View-Model. Sobald sich Eigenschaften im View-Model ändern, wird die View mittels der sogenannten Observer-Synchronisation benachrichtigt. Auf der anderen Seite werden die Eigenschaften der View-Model-Komponente direkt geändert, wenn zum Beispiel eine Benutzerinteraktion in der View erfolgt. Allgemein ist die View-Model-Komponente für die Verwaltung des Zustands der View und der Benutzerinteraktionen zuständig. Dabei verfügt sie über eine Referenz zum Model, damit sie auf die Anwendungsdaten und die Geschäftslogik zugreifen kann, ist aber ohne Kenntnis über jegliche View-Komponenten. Diese Verknüpfung funktioniert ausschließlich über das Prinzip der Datenbindung beziehungsweise der Observer-Synchronisation. Die Model-Komponente ist für die Datenspeicherung und Geschäftslogik verantwortlich und weiß nichts über die Existenz des View-Models und der View. (Syromiatnikov und Weyns, 2014)

Das Zusammenspiel zwischen der View- und der View-Model-Komponente ist bei diesem Entwurfsmuster besonders interessant. Microsoft hat für eine einfache Datensynchronisation mit loser Kopplung eine deklarative Datenbindung entwickelt. Sie erlaubt Eigenschaften von Elementen in der Benutzero-

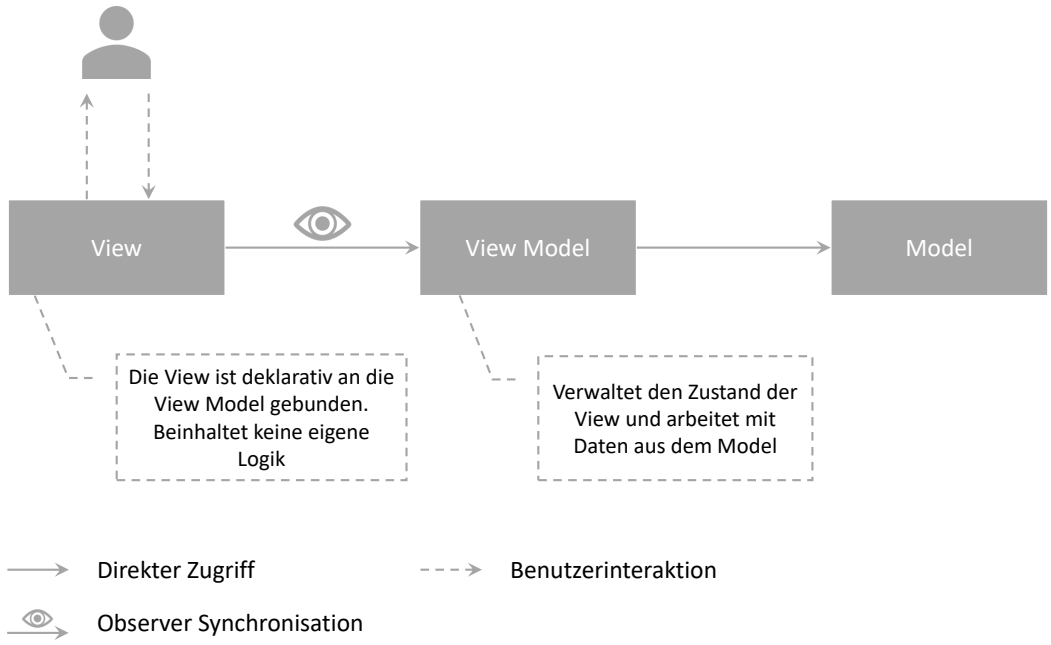

Abbildung 2.5: Microsoft MVVM Entwurfsmuster [Abbildung adaptiert von Syromiatnikov und Weyns (2014)]

berfläche mit Eigenschaften in der View-Model in einer deklarativen Art und Weise zu verbinden, ohne dass dafür expliziten Code in der View erstellt werden muss. Ändert sich das Element in der View, wird automatisch die korrespondierende Eigenschaft im View-Model ebenfalls aktualisiert und umgekehrt. Die Synchronisation wird dabei zur Gänze durch das Framework beziehungsweise deren immanenten Datenbindung ausgeführt. Der wesentliche Vorteil besteht darin, dass dadurch die Logik zwischen View und View-Model getrennt ist. Entwickler müssen daher nicht im Detail wissen, wie die View intern funktioniert, sondern können sich auf die Erstellung des Models und des View-Models fokussieren und die Erstellung der View können im Idealfall Designer mit geringer Programmiererfahrung übernehmen. (Syromiatnikov und Weyns, 2014)

Model-View-Presenter (MVP)

Das MVP-Entwurfsmuster ist ebenfalls ein Ansatz das MVC-Entwurfsmuster weiter zu verbessern. MVP verwendet ähnlich wie MVVM eine Observer-Synchronisation, indem eine Presenter-Komponente die View-Komponente überwacht, auf Benutzerinteraktionen reagiert und die View, wenn nötig, direkt aktualisiert. Innerhalb der MVP-Familie existieren unterschiedliche Abwandlungen des Entwurfmusters, in dieser Arbeit wird speziell das sogenannte Supervising-Presenter MVP betrachtet. (Syromiatnikov und Weyns, 2014)

Auch bei MVP besteht die Intention in der Separation der Geschäftslogik von der Präsentationslogik. Abbildung 2.6 zeigt schematisch die drei Komponenten mit jeweils dezidierten Aufgaben. Die Model-Komponente verwaltet die Anwendungsdaten und in manchen Fällen auch den Zustand der Präsentation. Die View-Komponente ist für die einfachen Zuordnungen zwischen der Benutzeroberfläche und den Daten des Models zuständig. Die Presenter-Komponente hingegen reagiert auf Benutzereingaben und beinhaltet die komplexe Präsentationslogik. (Syromiatnikov und Weyns, 2014)

MVP verfolgt einen eher Widget-basierten Ansatz für Benutzeroberflächen, mit dem Ziel das Verhalten des Entwurfsmusters klar und flexibel zu gestalten. Die direkte Interaktion zwischen Presenter und der View ermöglicht einen effizienten Zugriff auf Widgets innerhalb der View. Das Model bei MVP kann im Vergleich zu MVVM nicht nur Anwendungsdaten beinhalten, sondern auch Daten, welche für die korrekte Darstellung des Zustandes der View, benötigt werden. Es unterstützt dabei einen Benachrichtigungsmechanismus für die Überwachung der View. Die View-Komponente ist für die Darstellung der Daten und das Bearbeiten von einfachen Benutzereingaben zuständig. Dabei hat sie keinen direkten Zugriff auf die Presenter-Komponente, was zu einer losen Kopplung und damit zu einer besseren Einhaltung des SOC-Prinzips führt. Die Rolle der Presenter-Komponente besteht darin, die View und das Model miteinander zu verknüpfen, sodass sie zusammenarbeiten können. Der Presenter bearbeitet Benutzereingaben, aktualisiert das Model und die View und führt, wenn erforderlich, die Geschäftslogik aus. (Syromiatnikov und Weyns, 2014)

Die View reflektiert die Daten aus dem Model, führt jedoch keine Änderungen an diesen aus. Diese Aufgabe übernimmt der Presenter, wobei die View über keine direkte Referenz zur Presenter-Komponente verfügt, sondern einen Benachrichtigungsmechanismus dafür bereitstellt. Im Gegensatz dazu hat der Presenter sowohl eine Referenz zu View als auch zum Model. Der Presenter kann also die View direkt ändern, wobei dies nur in komplexen Fällen eingesetzt wird, wenn Daten, bevor sie dargestellt werden, transformiert werden müssen. (Syromiatnikov und Weyns, 2014)

Einerseits pflegt das MVP-Entwurfsmuster eine klare Trennung der Verantwortlichkeiten und entfernt die Logik aus der View. Andererseits verfügt die Presenter-Komponente über erhebliches Wissen über die View-Komponente. Dieser Aspekt führt zu Schwierigkeiten, wenn mehrere Views für dieselben Daten notwendig sind und hat auch generell negative Effekte auf die Testbarkeit der Anwendung. (Syromiatnikov und Weyns, 2014)

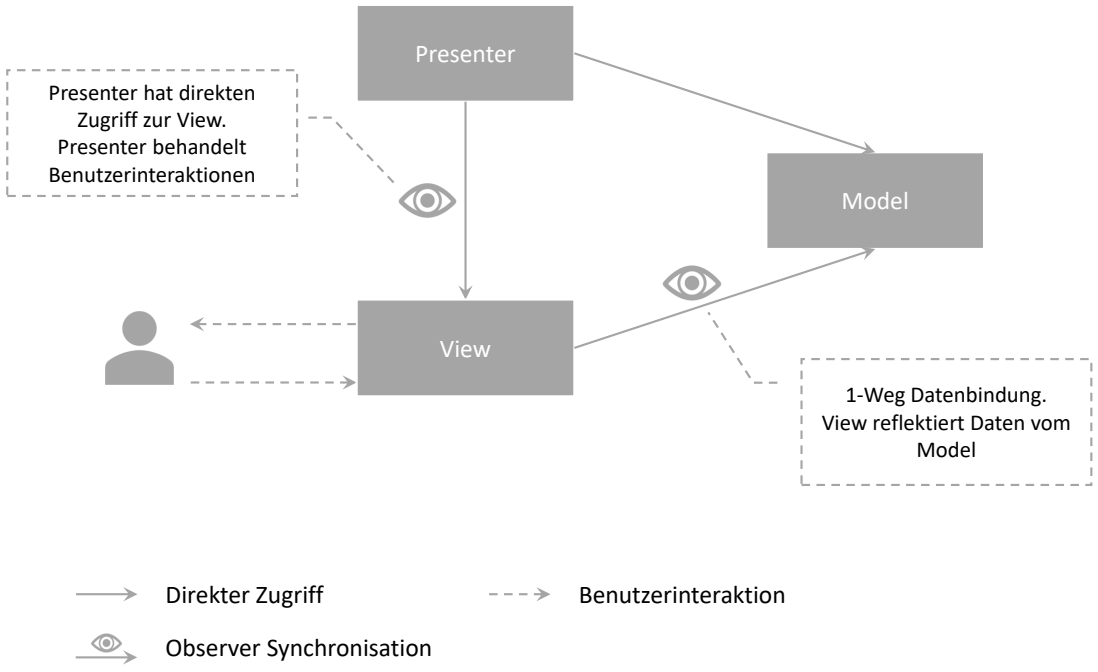

Abbildung 2.6: MVP Entwurfsmuster [Abbildung adaptiert von Syromiatnikov und Weyns (2014)]

2.5.3 Wahl der geeigneten Softwarearchitektur

In diesem Kapitel wird grob auf die Vor- und Nachteile der in den vorigen Kapiteln theoretisch betrachteten Microservice-Architektur 2.5.1 und einer klassischen, eher monolithischen Model-View-Architektur 2.5.2 eingegangen. Die Wahl einer geeigneten Softwarearchitektur kann in aller Regel nicht isoliert betrachtet werden, sondern hängt von vielen Faktoren ab, wie zum Beispiel der Problemstellung, der Art der Softwareanwendung, den involvierten Personen und nicht zuletzt auch der im jeweiligen Unternehmen gelebten Kultur, wie Software entwickelt wird.

Microservices sind eine Weiterentwicklung klassischer Service-orientierter Systeme. Die Architektur ermöglicht Anwendungen zu entwickeln, welche hoch skalierbar, sicher und leistungsfähig sind. Aufgrund der losen Kopplung einzelner Bereiche der Anwendung, die unabhängig voneinander aktualisiert werden können, ist der Einsatz verteilter Teams, welche die für ihre Zwecke passende Technologie einsetzen, möglich. Ein weiterer Vorteil, der sich aus der Art und Weise wie Microservices aufgebaut sind ergibt, ist die bessere Zuordnung der Verantwortlichkeiten. Dadurch ist häufig eine schnellere Problemlösung und damit eine höhere Verfügbarkeit der gesamten Anwendung erreichbar.

In monolithischen Architekturen, wie sie oft in Enterprise-Anwendungen eingesetzt werden, besteht prinzipiell kein Größenlimit, was häufig zu einer großen Codebasis führt. Mittels Model-View-Architekturen wird die Trennung zwischen Geschäftslogik und Präsentationslogik und damit die Einhaltung des SOC-Prinzips möglichst bewahrt. Dennoch besteht häufig eine enge Kopplung zwischen Modulen oder Services. Fehler betreffen deshalb oft die gesamte Anwendung und eine Skalierung erfordert meist das ganze System zu duplizieren. Bei der Entwicklung benötigen bereits kleine Änderungen eine komplett neue Erstellung der Anwendung, andererseits ist eine einfache Replikation der Entwicklungsumgebung realisierbar.

Die Microservice-Architektur setzt jedoch essentielle Bedingungen voraus, weshalb sie speziell für den Einsatz bei der Entwicklung von Software-Demonstratoren am Kompetenzzentrum ViF als nicht geeignet angesehen wird. Microservices bedingen wesentlich höhere Anfangskosten, so beispielsweise für die Aneignung der notwendigen Fähigkeiten, welche häufig bei anderen Architekturen bereits vorhanden sind. Verteilte Systeme erfordern zusätzliches Wissen und Erfahrungen im Umgang mit verteilten Transaktionen oder Reportings und beanspruchen ferner auch zusätzliche Test-Ressourcen für Latenztests und Tests der Ausfallsicherheit. Ein ausschlaggebender Aspekt für die Ablehnung von Microservices für die gegebene Situation, welche in dieser Arbeit behandelt wird, sind insbesondere die in frühen Phasen der Softwareentwicklung notwendigen Zusatzkosten um Microservices zu verwalten. Zudem entstehen erhöhte Anforderungen an die IT-Infrastruktur, welche bei Microservice-Anwendungen häufig an Cloud-Technologien ausgelagert werden, was aufgrund von Bedenken hinsichtlich der Datensicherheit bei vielen Kunden des Kompetenzzentrums ViF, nicht durchführbar ist. Schließlich wäre auch ein Kulturwechsel angesichts der Art und Weise, wie Microservice-Anwendungen entwickelt werden, insbesondere bei der Aufgliederung des Teams nach Geschäftsbereichen oder der Einsatz von Technologien zur Automatisierung der Entwicklung und Integration, erforderlich. Dieser Aspekt kann üblicherweise nur innerhalb eines länger dauernden Prozesses im Unternehmen und nicht von heute auf morgen erfolgen.

Auf Basis des Vorschlags keine Microservice-Architektur für die Entwicklung von Software-Demonstratoren beim ViF einzusetzen, werden im nächsten Kapitel 2.6 ausgewählte Technologien beziehungsweise Frameworks für die Unterstützung und Entwicklung von Webanwendungen, welche auf Model-View-Architekturen basieren, analysiert.

2.6 Ausgewählte webbasierte Technologien / Frameworks

In diesem Kapitel werden geeignete webbasierte Technologien beziehungsweise Frameworks für die Entwicklung von Software-Demonstratoren zuerst theoretisch betrachtet und anschließend wird versucht, anhand unterschiedlicher Faktoren eine Auswahl zu treffen, auf der die prototypische Implementierung aufbaut. Es handelt sich dabei lediglich um eine grobe Auswahl an Technologien, welche anhand der Randbedingungen der Arbeit ausgewählt worden sind, dies soll jedoch keine abschließende Aufstellung darstellen. Der Fokus wurde dabei speziell auf JavaScript-basierte Frameworks gelegt. Zuerst wird der sehr populäre MEAN-Stack in Kapitel 2.6.1 analysiert, welcher mehrere bewährte Technologien zusammenfasst. Anschließend wird eine Auswahl an sogenannten Full-Stack JavaScript-Frameworks in Kapitel 2.6.2 miteinander verglichen. Diese Frameworks implizieren bereits alle benötigten Technologien und vereinfachen damit die Entwicklung. Abschließend wird versucht, anhand der Analysen eine Technologie auszuwählen, auf derer später die Konzeptidee für die Umsetzung kontinuierlicher Entwicklungsprozesse bei der Entwicklung von Software-Demonstratoren aufbaut.

2.6.1 MEAN JavaScript-Stack

MEAN ist ein Akronym für MongoDB[7], Express[8], Angular[9] und Node[10]. Dabei wird MongoDB als Datenbank, Express als serverseitiges Framework, Angular als clientseitiges Framework und Node als JavaScript-basierte Server-Plattform für die Entwicklung von modernen Webapplikationen eingesetzt. Jede dieser Technologien wird von unterschiedlichen Teams entwickelt und es ist jeweils eine umfangreiche Community involviert, welche die Technologien betreuen und stetig weiterentwickeln. Es handelt sich dabei also um eine Zusammensetzung bewährter Technologien mit dem zugrundeliegenden Konzept, alle Bereiche einer Webanwendung mit der Programmiersprache JavaScript abzudecken und die Model-View-Architektur zu unterstützen oder teilweise sogar zu erwirken. Ein weiterer Vorteil besteht darin, dass eine Serialisierung und Deserialisierung von Datenstrukturen aufgrund der Verwendung von JSON-Objekten nicht mehr erforderlich ist. Nichtsdestotrotz liegt die Schwierigkeit darin, alle diese Technologien effektiv miteinander zu verbinden, damit später keine Probleme zum Beispiel bezüglich der Skalierbarkeit auftreten. (Haviv, 2014)

Für MEAN existieren zwei Implementationen: einerseits MEAN.IO[11] und MEAN.JS[12]. Beide Lösungen wurden von Amos Q. Haviv initiiert, wobei MEAN.JS eine Abspaltung der ersten Open-Source-Lösung ist und er sich aktuell auch dort aktiv betätigt und das Framework unterstützt. (Haviv, 2014)

Wie bereits erwähnt handelt es sich bei MEAN lediglich um eine Zusammensetzung von bewährten Technologien. Viele Schwierigkeiten und offene Fragen müssen bei der Entwicklung trotz Einsatz des MEAN-Stack berücksichtigt werden:

- Wie können alle Technologien miteinander effektiv verbunden werden?
- Node verfügt über ein umfangreiches Ökosystem an Modulen, welche Module werden benötigt?
- Wie kann die Model-View-Architektur durchgesetzt werden, obwohl JavaScript generell keine Architektur erzwingen kann?
- Bei JSON handelt es sich um eine Datenstruktur ohne Schema. Wie können die Daten trotzdem modelliert werden?

[7] MongoDB - NoSQL-Datenbank - https://www.mongodb.com/
[8] Express - Serverseitiges Web-Framework für Node - http://expressjs.com/
[9] Angular - Clientseitiges Web-Framework - https://angular.io/
[10] Node - JavaScript-basierte Server-Plattform - https://nodejs.org/
[11] MEAN.IO Implementierung - http://mean.io/
[12] MEAN.JS Implementierung - http://meanjs.org/

- Wie kann die Authentifizierung von Benutzern realisiert werden?
- Wie können Echtzeit-Anwendungen entwickelt werden?
- Wie kann eine MEAN-Applikation getestet werden?

(Haviv, 2014)

Diese Aspekte können alle innerhalb des MEAN-Stacks gelöst werden, erfordern jedoch häufig über fundiertes Wissen und Erfahrung in verschiedensten Teilbereichen unterschiedlicher Technologien. Sowohl MEAN.JS als auch MEAN.IO können als Full-Stack JavaScript-Frameworks betrachtet werden, weil sie nützliche Tools für die Erstellung von MEAN-Applikationen bereitstellen. Dennoch ist die Unterstützung im Vergleich zu anderen Frameworks begrenzt und der MEAN-Stack an sich liefert ein Paket an Technologien, welche nicht automatisch nahtlos miteinander verknüpft sind. Aus diesem Grund werden im nächsten Kapitel 2.6.2 einige ausgewählte Full-Stack JavaScript-Frameworks verglichen, welche ebenso auf denselben oder ähnlichen Technologien aufbauen, aber die Entwicklung von Webanwendungen noch weiter zu vereinfachen und zu optimieren versuchen.

2.6.2 Full-Stack JavaScript-Frameworks

Unter Full-Stack JavaScript-Frameworks werden integrierte Lösungen verstanden, welche es ermöglichen, bei Webapplikation sowohl die serverseitige- als auch die clientseitige-Implementation zur Gänze in JavaScript zu entwickeln. In diesem Kapitel werden einige ausgewählte Full-Stack JavaScript-Frameworks miteinander verglichen. Die Anzahl an solchen Frameworks, welche wirklich versuchen alle Bereiche bei der Entwicklung einer modernen Webapplikation abzudecken, ist derzeit noch überschaubar, die Menge an neuen Frameworks nimmt jedoch stetig zu. Dennoch soll auch in diesem Vergleich nicht der Anspruch auf Vollständigkeit erhoben, sondern lediglich ein grober Überblick vermittelt werden.

Die Analyse wurde im Juli 2016 durchgeführt und es wurden folgende ausgewählte Full-Stack JavaScript-Frameworks betrachtet:

- Meteor[13]
- Derby[14]
- wakanda[15]
- CleverStack[16]

Ein essentielles Kriterium bei der Wahl eines geeigneten Frameworks war aufgrund der Randbedingungen der Arbeit, dass sowohl das Framework als auch die intern verwendeten Technologien als Open-Source-Software zur Verfügung stehen. Diese Bedingung erfüllen, wie in Tabelle 2.4 ersichtlich, alle ausgewählten Frameworks. Mit Ausnahme von wakanda wurden alle Frameworks von Beginn an als Open-Source-Software konzipiert. Das wakanda-Framework wurde anfänglich vom Unternehmen 4D entwickelt und erst zu einem späteren Zeitpunkt als Open-Source-Lösung zur Verfügung gestellt. Alle vier Frameworks werden, im Fall von wakanda zumindest in einer Edition, in der freien MIT-Lizenz[17] angeboten.

[13] Meteor - https://www.meteor.com/
[14] Derby - http://derbyjs.com/
[15] wakanda - https://wakanda.github.io/
[16] CleverStack - http://cleverstack.io/developer/
[17] MIT-Lizenz - https://opensource.org/licenses/MIT/

	Meteor	Derby	wakanda	CleverStack
Open-Source	✓	✓	✓ (anfänglich 4D)	✓
Lizenzmodell	MIT Lizenz	MIT Lizenz	MIT Lizenz (Community Edition)	MIT Lizenz

Tabelle 2.4: Lizenz Vergleich an Full-Stack JavaScript-Frameworks (Stand Juli 2016)

In Tabelle 2.5 werden die verschiedenen Frameworks von einem technischen Blickwinkel aus betrachtet. Dabei unterscheiden sich vor allem Meteor und Derby, welche ein isomorphes Paradigma einsetzen, von den restlichen Frameworks wakanda und CleverStack, welche einen eher traditionellen Ansatz mit einer klaren Trennung zwischen Server und Client verfolgen. Bei isomorphen JavaScript-Frameworks kann der gleiche Code sowohl auf dem Server als auch auf dem Client ausgeführt werden. Dieses Paradigma eröffnet eine Vielzahl an Möglichkeiten, welche in Kapitel 3.1.1 später genauer erläutert werden. Die Interaktion zwischen den Frameworks und dem Benutzer erfolgt, mit Ausnahme von wakanda, mittels der Kommandozeile. Wakanda ist diesbezüglich das einzige Framework, welches eine erweiterte Entwicklungsumgebung (IDE) mit einem integrierten Code Editor für die Entwicklung bereitstellt. Bei den intern eingesetzten Technologien bauen alle Frameworks auf die serverseitige JavaScript-Plattform Node, welche die Ausführung von JavaScript-Code auf dem Server ermöglicht. Eine detaillierte Betrachtung der Funktionsweise von Node findet im Kapitel 3.2.1 statt. CleverStack und wakanda unterstützen diverse Datenbanken, demgegenüber unterstützen Meteor und Derby, auch aufgrund des isomorphen Paradigmas, zum Zeitpunkt der Analyse lediglich MongoDB. Dabei handelt es sich um keine relationale, sondern um eine flexible, skalierbare, dokumentenorientierte Datenbank. MongoDB wird ausführlich im Kapitel 3.2.3 erläutert. Als clientseitiges Framework vertrauen die meisten Full-Stack Frameworks auf Angular, wobei zum Beispiel in Meteor auch React oder Blaze eingesetzt werden können.

	Meteor	Derby	wakanda	CleverStack
Framework-Paradigma	isomorph	isomorph	traditionell	traditionell
Benutzerinteraktion	Kommandozeile	Kommandozeile	Wakanda Studio Code Editor Kommandozeile für Server	Kommandozeile
Technologie	Node MongoDB Angular / React / Blaze	Node MongoDB Redis	Node Angular Ionic Cordova	Node Angular NPM / Bower Grunt

Tabelle 2.5: Technischer Vergleich an Full-Stack JavaScript-Frameworks (Stand Juli 2016)

Ein ebenso bedeutsames Kriterium für die Wahl eines Frameworks ist die Reichweite und die Akzeptanz innerhalb der Entwickler-Community. Dazu wurden die Bewertungen, die Forks (heruntergeladener Code) und die Bookmarks auf GitHub[18] aller ausgewählten Frameworks in Tabelle 2.6 miteinander verglichen. Dabei zeigt sich deutlich, dass Meteor in diesem Punkt führend und alle anderen Frameworks weit abgeschlagen sind. Ein ähnliches Bild ergibt sich bei der Betrachtung der Anzahl an Fragen auf Stackoverflow[19] in Tabelle 2.7. Bei Stackoverflow handelt es sich um eine populäre Internetplattform, auf welcher Benutzer Fragen zum Thema Softwareentwicklung stellen können, welche wiederum von

[18] GitHub - Entwickler Plattform - https://github.com/

[19] Stackoverflow - Internetplattform für Fragen zum Thema Softwareentwicklung - https://stackoverflow.com/

anderen Benutzern versucht werden zu beantworten. Auch bei dieser Betrachtung scheint Meteor die ungleich größere oder zumindest aktivere Community hinter sich vereinen zu können.

	Meteor	Derby	wakanda	CleverStack
Bewertungen	> 34000	> 4000	> 28	> 200
Forks	> 4200	> 230	> 30	> 35
Bookmarks	> 1900	> 190	> 38	> 20

Tabelle 2.6: GitHub Vergleich an Full-Stack JavaScript-Frameworks (Stand Juli 2016)

	Meteor	Derby	wakanda	CleverStack
Anzahl an Fragen	> 22100	> 1800	> 77	-

Tabelle 2.7: Stackoverflow Vergleich an Full-Stack JavaScript-Frameworks (Stand Juli 2016)

Zusammenfassend unterscheiden sich die isomorphen Full-Stack JavaScript-Frameworks Meteor und Derby technisch nur gering, allerdings verfügt Meteor über die wesentlich größere Reichweite und Akzeptanz in der Entwickler-Community. Ähnlich verhält es sich bei den traditionellen Full-Stack Frameworks wakanda und CleverStack, wobei wakanda als einziges Framework in der Auswahl anstelle einer einfachen Kommandozeile auch eine integrierte Entwicklungsumgebung in Form des 'Wakanda Studios' bereitstellt. Dieser Aspekt ist insofern in Verbindung mit EUSE von essentieller Bedeutung. Leider beschränkt sich die Unterstützung nur auf die serverseitige Implementierung und das wakanda-Framework bietet keine zusätzlichen Tools für die clientseitige Entwicklung mittels Angular. Abschließend lässt sich festhalten, dass Meteor in dieser Auswahl an Full-Stack JavaScript-Frameworks subjektiv nach empirischer Überprüfung und auch nach objektiven Kriterien, wie der Reichweite und Unterstützung auf GitHub oder Stackoverflow, das geeignetste Full-Stack Framework zu sein scheint. Allerdings kann auch Meteor nicht zur Gänze alle Anforderungen, insbesondere hinsichtlich EUSE abdecken. Die Interaktion mittels Kommandozeile erschwert in den meisten Fällen die aktive Integration von End-Usern in den Entwicklungsprozess und muss daher im späteren Konzeptvorschlag in Kapitel 2.7 berücksichtigt werden.

2.6.3 Wahl der geeigneten Technologie / Framework

Nachdem in den vorherigen Kapiteln der MEAN-Stack (2.6.1) und Full-Stack JavaScript-Frameworks (2.6.2) betrachtet wurden, wird in diesem Kapitel versucht, kurz auf die Vor- und Nachteile der jeweiligen Technologien beziehungsweise Frameworks einzugehen und abschließend ein Vorschlag für ein bestimmtes Framework zu unterbreiten, auf dem die prototypische Implementation im späteren Teil der Arbeit aufbaut.

Beim MEAN-Stack handelt es sich um eine Zusammensetzung aus bewährten Technologien, ansonsten wird jedoch keine zusätzliche Unterstützung für die Entwicklung bereitgestellt. Es müssen dabei MongoDB manuell mit Node und Express mit Angular verbunden werden. Zudem liegt es in der Verantwortung der Entwickler, REST-Endpunkte serverseitig zu erstellen und clientseitig zu konsumieren. Dieser Aspekt kann jedoch auch als Vorteil betrachtet werden, weil er Entwicklern weniger Abhängigkeiten, mehr Freiraum und dadurch mehr Flexibilität ermöglicht. Bei der Entwicklung von Software-Demonstratoren sind jedoch unterschiedliche Stakeholder mit vielfältigen Qualifikationen hinsichtlich Softwareentwicklung involviert, für welche diese Kriterien nicht erstrangig sind. (Greussing und Zoier, 2016)

Full-Stack JavaScript-Frameworks bieten zusätzliche Funktionen wie zum Beispiel ein integriertes Paket-Management, Session-Management, Unterstützung für mobile Applikationen, reaktive Templates oder

das sogenannte Hot-Code-Reload, welches dafür sorgt, dass genau der Teil der Anwendung automatisch neu geladen wird, indem eine Änderung im Code stattgefunden hat. Darüber hinaus stellen integrierte Lösungen bereits die komplette Kette an Tools zur Verfügung, welche für die Erstellung einer Anwendung benötigt werden. Dazu gehören die entsprechenden Compiler für Typescript, CSS-Precompiler wie zum Beispiel Sass[20] oder ein Tool für die Komprimierung und Zusammenführung von JavaScript- und Stylesheet-Dateien. Für diese Aufgaben werden im MEAN-Stack sogenannte Task-Runner wie Grunt oder Gulp eingesetzt, welche jedoch eine immanente Komplexität beinhalten. (Greussing und Zoier, 2016)

Aufgrund des isomorphen Paradigmas, welches sicherstellt, dass der gleiche Code sowohl auf dem Client als auch auf dem Server funktioniert, können Webanwendungen auf eine ähnliche Art und Weise implementiert werden wie klassische Desktopanwendungen. Dadurch wird die Komplexität in vielen Fällen verringert und involvierte Teams haben meist ein besseres Verständnis über den Code des gesamten Projektes. (Greussing und Zoier, 2016)

Abschließend lässt sich festhalten, dass vermutlich kein Framework alle Anforderung zur Gänze abdecken kann. Aufgrund der vielfältigen Stakeholder mit unterschiedlichsten Qualifikationen hinsichtlich Softwareentwicklung werden zusätzlich erhöhte Anforderungen an ein zugrundeliegendes Framework und unterstützende Plattformen gestellt. Die hohe Dynamik, die mittlerweile überwältigende Auswahl an Technologien und die strikte Trennung zwischen Server- und Client-Entwicklung in der klassischen Webentwicklung erhöhen jedoch meist die Komplexität solcher Projekte und erfordern fundierte Erfahrung in den entsprechenden Frameworks beziehungsweise Technologien. Dieser Aspekt verhindert meist eine effektive Integration von End-Usern in den aktiven Entwicklungsprozess. Für eine wirksame Einbindung von End-Usern in die Entwicklung von webbasierten Software-Demonstratoren scheint ein Framework erfolgversprechend, welches sowohl die Server- als auch die Client-Umgebung berücksichtigt, möglichst viele Design- und Technologieentscheidungen abstrahiert und damit die Komplexität minimiert. Das ist auch der vordringliche Grund, weshalb in weiterer Folge in dieser Arbeit ein Full-Stack JavaScript-Framework und in dieser Familie - nach der Analyse in Kapitel 2.6.2 - das Meteor Framework eingesetzt wird.

2.7 Zusammenfassung und Konzeptidee

Die Entwicklung von Software-Demonstratoren zur optimalen, digitalen Informationsversorgung für Ingenieure in der Automobilindustrie erfordert eine gesamtheitliche Betrachtung vieler Bereiche in der Softwareentwicklung. Aus diesem Grund wurde zuerst in Kapitel 2.4 das Thema EUSE theoretisch betrachtet und deren Bedeutsamkeit hervorgehoben. Anschließend wurden ausgewählte Softwarearchitekturen in Kapitel 2.5 mit dem Ergebnis analysiert, dass eine klassische Architektur aus der Familie der Model-View-Architekturmustern für die prototypische Implementierung eine geeignete Lösung darstellen könnte. Abschließend wurden ausgewählte Frameworks beziehungsweise Technologien in Kapitel 2.6 miteinander verglichen und der Vorschlag verbreitet, das isomorphe Full-Stack JavaScript-Framework Meteor als Technologie für die weitere Vorgehensweise zu verwenden.

Die Problematik bei Meteor besteht darin, dass die Benutzerinteraktion mit dem Framework auf einer eher rudimentären Ebene mithilfe der Kommandozeile erfolgt und dieser Aspekt die Integration von End-Usern in die Softwareentwicklung erschwert. Die Konzeptidee dieser Arbeit schlägt deshalb vor, eine Abstraktionsebene über Meteor zu implementieren, welche eine webbasierte, benutzerfreundliche und integrierte Entwicklungsumgebung für Meteor repräsentieren soll. Abbildung 2.7 illustriert anschaulich das Konzept mit den unterschiedlichen Abstraktionsebenen auf denen schlussendlich ein Projekt

[20] Sass - CSS-Precompiler - http://sass-lang.com/

aufgebaut wird. Je höher die Abstraktionsebene, desto größer die Abstraktion und die Benutzerfreundlichkeit für involvierte End-User, aber desto geringer die Komplexität, Flexibilität und Funktionalität für professionelle Softwareentwickler. Die verschiedenen Ebenen stellen gleichzeitig auch unterschiedliche Projektphasen innerhalb eines Forschungsprojekts beim ViF dar. Der Lösungsansatz muss daher sicherstellen, dass jederzeit von einer höheren Ebene auf eine niedrigere Ebene mit geringerer Abstraktion gewechselt werden kann ohne, dass dabei ein Technologiewechsel notwendig ist.

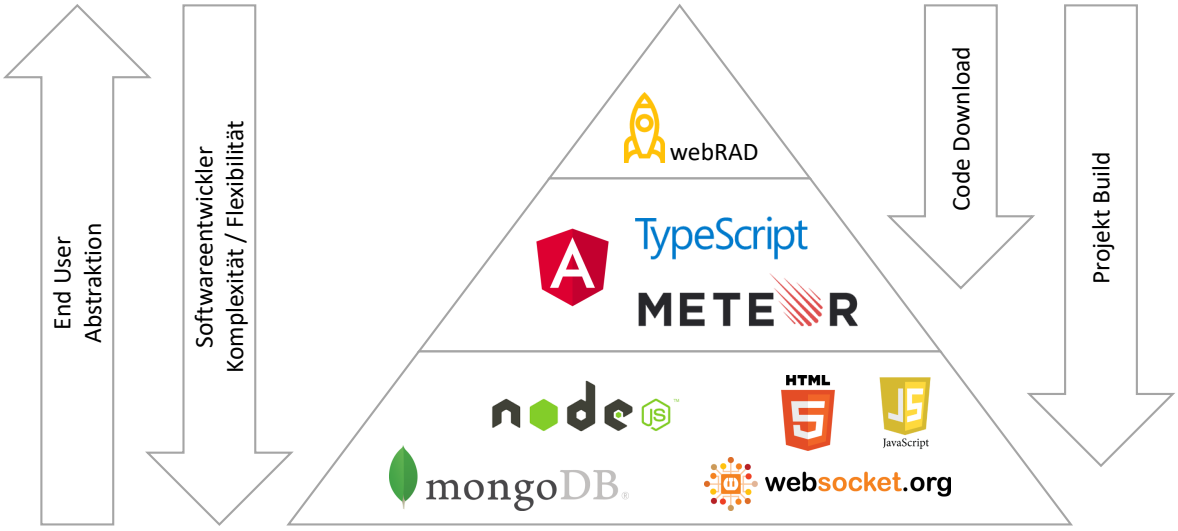

Abbildung 2.7: Konzeptidee für die Entwicklung von Software-Demonstratoren

Während der ersten Projektphase befindet sich die Entwicklung von Software-Demonstratoren auf der höchsten Abstraktionsebene. Diese Phase ist gekennzeichnet durch eine enge Einbindung von Fahrzeugexperten in den Softwareentwicklungsprozess. Idealerweise können dabei Experten in der Automobilindustrie - selbständig oder begleitend durch Fahrzeugexperten beim ViF - Aktivitäten bei der Entwicklung von Software-Demonstratoren übernehmen. Meteor stellt für diesen Zweck jedoch keine geeignete Entwicklungsumgebung zur Verfügung. Es ist daher Teil dieser Arbeit eine prototypische Implementierung eines webbasierten Editors mit dem Arbeitstitel 'webRAD', welcher auf Meteor und Angular aufbaut, zu entwickeln. Kapitel 5 befasst sich deshalb ausführlich mit dem Aufbau und der Funktionsweise von webRAD.

Ein Projekt, welches mittels webRAD erstellt wird, baut intern maßgeblich auf Meteor und Angular auf. Das bedeutet, dass in späteren Phasen eines Projektes, in welchen häufig zusätzliche Funktionalität gefordert wird, jederzeit der bereits generierte Code ohne Abhängigkeiten zu webRAD weiterverwendet werden kann. Dieser Aspekt ermöglicht einen durchgängigen Entwicklungsprozess von Software-Demonstratoren mit anfänglich enger Einbindung von End-Usern beziehungsweise Fachexperten und zusätzlich hoher Flexibilität und Funktionalität für professionelle Softwareentwickler in späten Projektphasen.

Die unterste Abstraktionsebene repräsentieren die in Meteor und Angular inhärenten Technologien, wie zum Beispiel Node, MongoDB, WebSockets, JavaScript oder HTML5. Meteor-Applikationen können als reine Node-Applikationen auch direkt aus webRAD bereitgestellt werden, welche unabhängig von Meteor weiterentwickelt werden können.

Die in diesem Kapitel vorgestellte Konzeptidee erlaubt die kontinuierliche Entwicklung von Software-Demonstratoren über mehrere Projektphasen eines Forschungsprojektes beim ViF hinweg. Diese De-

monstratoren können anfänglich durch Fachexperten in der Automobilindustrie mittels webRAD mitentwickelt werden. In späteren Projektphasen können komplexere Anwendungsfälle durch Softwareentwickler beim ViF umgesetzt werden. Dabei können sie auf Ergebnisse aus früheren Projektphasen aufbauen, weil keine grundlegend neue Technologie eingesetzt werden muss, sondern lediglich auf einer anderen Abstraktionsebene weiterentwickelt wird. Die Konzeptidee baut im Wesentlichen auf dem isomorphen Full-Stack JavaScript-Framework Meteor (Kapitel 3) mit dem clientseitigen Framework Angular (Kapitel 4) auf. Die nächsten beiden Kapitel widmen sich daher ausführlich diesen beiden Technologien, bevor anschließend die prototypische Implementierung von webRAD in Kapitel 5 detailliert erläutert wird.

Kapitel 3

Einführung in das Meteor Framework

“ Simplicity is prerequisite for reliability. ”

[Dijkstra, 1975]

Edsger Dijkstra, ein renommierter Informatiker, formuliert mit seinem Zitat zutreffend die Philosophie, welche Meteor zugrunde liegt. Meteor ist in vielerlei Hinsicht ein revolutionäres Full-Stack JavaScript-Framework zur Erstellung von modernen Webapplikationen. Es vereinfacht die Erstellung von interaktiven, webbasierten Anwendungen, welche auf dem Model-View-ViewModel (MVVM)-Entwurfsmuster basieren. Das erste initiale Release von Meteor wurde Anfang 2012 als Open-Source-Lösung auf GitHub veröffentlicht und ist auf sieben grundlegenden Prinzipien, wie in Abbildung 3.1 schematisch dargestellt, aufgebaut:

1. One Language - Isomorphic JavaScript
   Meteor erlaubt es die komplette Applikation, sowohl Server als auch Client, in JavaScript zu schreiben.

2. Data on the Wire
   Der Server sendet Daten und kein HTML. Der Client ist für die Interpretation und Anzeige der Daten verantwortlich.

3. Latency-Compensation
   Der Client kann simulierte Daten vorab abrufen, sodass der Eindruck entsteht, Server-Methoden werden lokal auf dem Client ausgeführt.

4. Database Everywhere
   Mittels gleicher API erfolgt ein direkter Zugriff auf die Datenbank. Auf dem Client wird der Datenbankzugriff simuliert.

5. Full-Stack Reactivity
   Die Benutzeroberfläche reflektiert zu jeder Zeit den wirklichen Zustand der Anwendung wieder.

6. Embrace the Ecosystem
   Meteor und seine Komponenten sind frei zugänglich. Das Framework interagiert mit bestehenden Technologien, anstatt sie zu ersetzen.

7. Simplicity Equals Productivity
   Eine einfache und saubere API erhöht die Produktivität und Zuverlässigkeit.

(Coleman und Nguyen, 2016)

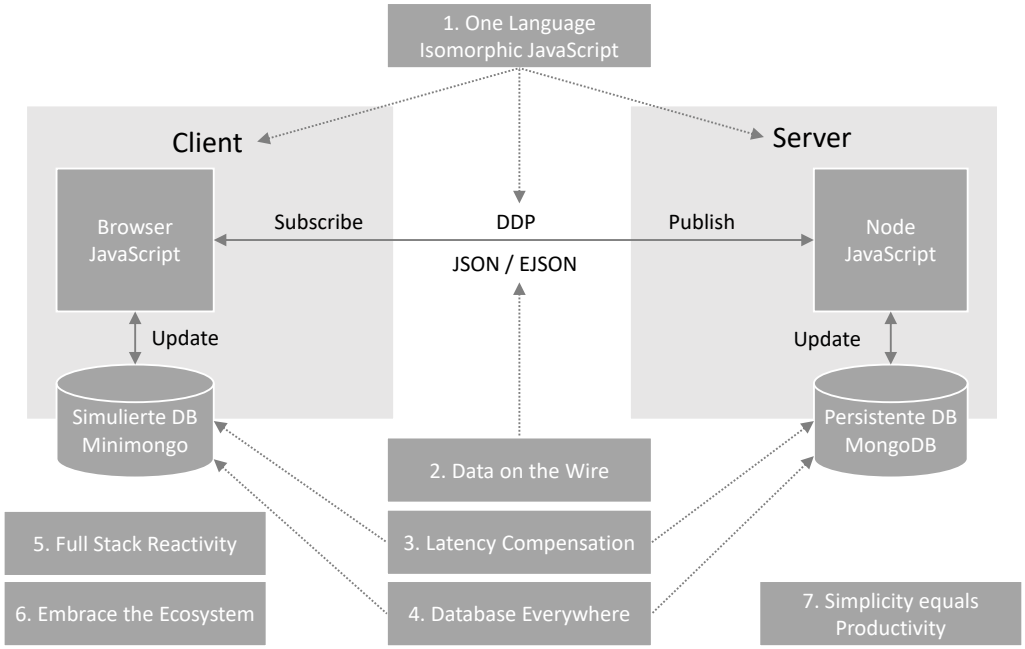

Abbildung 3.1: Sieben grundlegende Prinzipien, auf denen Meteor aufgebaut ist [Abbildung adaptiert von Wehrle (2015)]

3.1 Grundlegende Prinzipien von Meteor

Dieses Kapitel widmet sich ausgewählten, elementaren Prinzipien, auf denen Meteor aufgebaut ist und analysiert, wie diese Prinzipien im Framework integriert sind.

3.1.1 One Language - Isomorphic JavaScript

Das Wort isomorph stammt aus dem Griechischen und bedeutet ’iso’ (gleich) und ’morph’ (Form, Gestalt). Allgemein beschreibt der Isomorphismus, dass, wenn dieselbe Entität von zwei unterschiedlichen Kontexten betrachtet wird, trotzdem ein gleiches Resultat erwartet werden darf. Im Zusammenhang mit JavaScript-Frameworks sind die Kontexte auf der einen Seite der Server und auf der anderen Seite der Client. Der Ausdruck ’Isomorphismus’ ist hauptsächlich in der Mathematik gebräuchlich, wird gegenwärtig aber auch in der Entwicklung von Webanwendungen für die Beschreibung von Code, welcher zwischen Server und Client geteilt wird, angewendet. (Bédard, 2015)

Isomorphic JavaScript bedeutet, dass jede Zeile Code in JavaScript geschrieben werden kann und sowohl auf dem Server als auch auf dem Client ausführbar ist. Dieses Prinzip eröffnet eine Vielzahl an neuen Möglichkeiten und führt zu einer einfacheren Wartbarkeit, einer effektiveren Suchmaschinenoptimierung und einer besseren Performance. Meteor verwendet Node als serverseitige JavaScript-Plattform, welche im Kapitel 3.2.1 im Detail beschrieben wird, und bietet darüber hinaus auch den gleichen Zugriff auf die Datenbank-API und erfüllt dadurch die Anforderungen an ein vollständiges isomorphes JavaScript-Framework. (Brehm, 2013)

In den Anfängen der Web-Entwicklung waren Browser noch nicht wirklich leistungsfähig. Ihre Aufgabe bestand darin, Anfragen an den Server zu senden, welcher daraus eine HTML-Seite generierte und sie an den Browser zurück übermittelte. JavaScript erlaubte es, diese statischen Seiten dynamischer zu gestalten, die Möglichkeiten waren aber zu Beginn meist sehr begrenzt. Die Leistungssteigerung der Computer und die Weiterentwicklung der Web-Standards und damit der Browser ermöglichte es, aufwendigere und

dynamischere Webapplikationen zu erstellen, welche vorher nur als native Anwendungen denkbar waren. Die neuen Möglichkeiten führten dazu, dass immer mehr Aufgaben und Logik direkt im Browser beim Client implementiert wurden. Es wurden clientseitige Model-View-Controller (MVC)-Frameworks wie Angular, Backone oder Ember entwickelt, welche die Erstellung von Single-Page-Applikation (SPA) erheblich erleichterten. Abbildung 3.2 veranschaulicht schematisch eine typische clientseitige MVC-Architektur solcher modernen Web-Anwendungen. (Brehm, 2013)

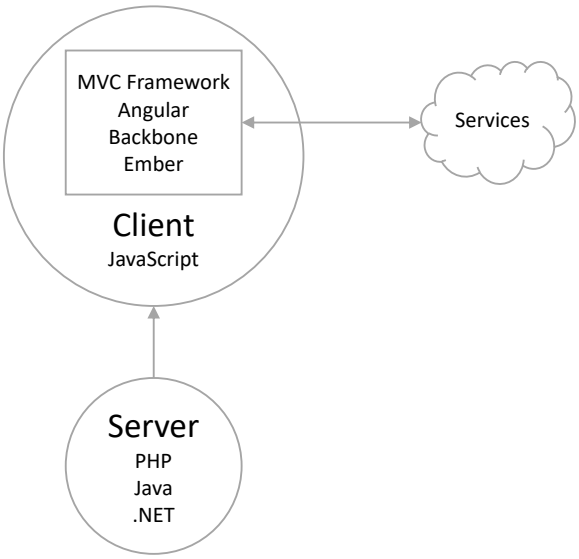

Abbildung 3.2: Clientseitige MVC-Architektur ohne isomorphes JavaScript [Abbildung adaptiert von Brehm (2013)]

Der Server kann dabei in einer beliebigen Programmiersprache wie PHP, Java oder .NET geschrieben sein. Er ist im Wesentlichen dafür zuständig, das initiale Grundgerüst mit dem clientseitigen Framework an den Client zu senden. Sobald die Applikation initialisiert wurde, übernimmt meist der Client die Kontrolle, indem er dynamisch Inhalte über Services ladet und sie dann im Browser darstellt. Für den Benutzer hat diese Architektur den Vorteil, dass sie eine schnelle Navigation ermöglicht ohne jedes Mal die Seiten neu laden zu müssen. Richtig angewendet kann auch die Anforderung eines Offline-Modus mittels dieser Architektur realisiert werden. Auf der anderen Seite profitieren Entwickler von der strikten Trennung zwischen Client und Server, welche es erleichtert das wichtige Softwareentwicklungs-Prinzip Separation of Concern (SOC), also die Trennung der Zuständigkeiten, richtig anzuwenden. (Brehm, 2013)

Der Lösungsansatz eines intelligenten Clients und eines hauptsächlich initial erforderlichen Servers hat jedoch auch Nachteile und ist deshalb nicht für alle Anwendungsfälle die geeignete Lösung. Die Search-Engine-Optimization (SEO) funktioniert bei Applikationen, welche zu einem überwiegenden Teil nur auf dem Client laufen, üblicherweise nicht gut. Suchmaschinen-Crawler wurden in den letzten Jahren immer leistungsfähiger und können mittlerweile auch Single-Page-Applikationen besser indizieren. Nichtsdestotrotz bleibt diese Architektur für Suchmaschinen problematisch, dieser Aspekt sollte daher bereits in der Anforderungsanalyse an die Applikation mitberücksichtigt werden. Die strikte Trennung zwischen Server und Client hilft, die Trennung der Zuständigkeiten einzuhalten, führt jedoch oft zwangsläufig zur Verletzung des Don't Repeat Yourself (DRY)-Prinzips[1]. Häufig lässt es sich in einer clientseitigen MVC-Architektur nicht vermeiden, dass Code, wie zum Beispiel die Validierung von Eingabedaten oder die Formatierung von Daten, sowohl auf dem Client als auch auf dem Server implementiert werden muss. Zusätzlich werden diese gleichen Codeteile meist auch noch in unterschiedlichen Programmiersprachen

[1] Don't Repeat Yourself (DRY)-Prinzip - http://programmer.97things.oreilly.com/wiki/index.php/Don't_Repeat_Yourself

geschrieben. Darunter leidet insbesondere bei größeren und komplexeren Anwendungen die Wartbarkeit des gesamten Codes. Ein weiterer Nachteil besteht darin, dass bei Single-Page-Applikationen Benutzer eine längere Wartezeit zu erdulden haben, bis die erste, initiale Seite geladen wird. Dieses Problem entsteht angesichts der Tatsache, dass der Server keine fertigen HTML-Seite an den Client sendet, sondern der Client selber HTML mittels JavaScript generiert. Dadurch bilden sich ein paar kritische Sekunden an Wartezeit, in welcher dem Benutzer lediglich eine leere Seite oder eine Forschrittsanzeige angezeigt wird. Studien haben gezeigt, dass langsame Webseiten gravierend die Benutzerfreundlichkeit und dadurch auch den Umsatz verschlechtern. Dabei haben speziell beim Laden der ersten Seite bereits wenige Millisekunden drastische Auswirkungen. (Brehm, 2013)

Eine vielversprechende Lösung für diese Probleme scheint daher ein hybrider Ansatz mittels isomorphes JavaScript, wie in Abbildung 3.3 dargestellt, zu sein. Das Ziel dieser Architektur besteht darin, dass der Server vollständige HTML-Seiten an den Client liefern kann und damit die SEO-Problematik und die Wartezeit bis die erste Seite geladen wird verringert. Gleichzeitig wird die Flexibilität und Geschwindigkeit in der Navigation von clientseitigen MVC-Frameworks beibehalten. (Brehm, 2013)

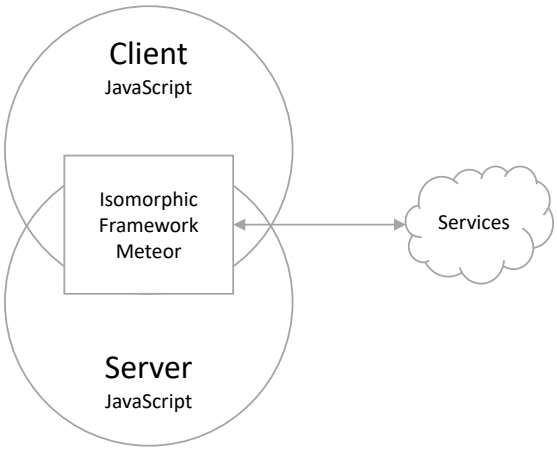

Abbildung 3.3: Client- und serverseitige MVC-Architektur mittels isomorphem JavaScript [Abbildung adaptiert von Brehm (2013)]

Meteor verfolgt das Prinzip von isomorphem JavaScript von Grund auf und verwischt dabei die Grenzen zwischen Server und Client. Beide Umgebungen verwenden die gleiche Programmiersprache - JavaScript - und teilen gemeinsamen Code. JavaScript kann schon länger im Browser verwendet werden und dank Node (siehe Kapitel 3.2.1) ist dies auch auf dem Server möglich. Meteor schafft eine Abstraktionsebene über Client und Server und ermöglicht, dass beide Seiten nahtlos miteinander kommunizieren können. Das Framework erlaubt dabei auf zwei verschiedene Arten zu kontrollieren, in welcher Umgebung der jeweilige Code ausgeführt werden soll. Dies geschieht einerseits über die Dateistruktur, indem die Code-Dateien in speziellen Verzeichnissen (/server beziehungsweise /client) abgelegt werden und andererseits über die booleschen Eigenschaften 'Meteor.isServer' und 'Meteor.isClient'. JavaScript-Befehle, welche nur im Kontext eines Browsers Sinn machen, wie zum Beispiel der 'alert' Befehl, sollten in einem isomorphen JavaScript-Framework nur clientseitig eingesetzt werden. Collections sind ein typisches Beispiel für Code, welcher in beiden Umgebungen ausgeführt wird. In Meteor kann mittels Collections auf die MongoDB Datenbank zugegriffen werden. Dabei handelt es sich um die Deklaration von Models zur Speicherung von Daten, analog zu traditionellen Object-Relation-Mapper (ORM)-zentrierten Frameworks. MongoDB und Collections werden im Kapitel 3.2.3 im Detail beschrieben. (Greif, 2015)

Die wesentlichen Vorteile einer isomorphen JavaScript-Architektur liegen in der Verwendung einer einzigen Programmiersprache zur Entwicklung einer gesamten Webapplikation und in der Vermeidung von

duplizierter Logik, indem Code zwischen Server und Client geteilt werden kann. Diese Vorteile bringen jedoch auch gleichzeitig Herausforderungen mit sich. Die Architektur weicht teilweise erheblich von den bisher üblichen Architekturen in der Web-Entwicklung ab und hat daher eine größere Lernkurve für Entwickler, welche zuvor noch nie dieses Prinzip angewendet haben. Eine weitere Herausforderung betrifft das Debuggen von solchen isomorphen Anwendungen. Der Grund dafür ist, dass meist zuerst analysiert werden muss, wo der Code ausgeführt wird und dementsprechend unterschiedliche Debug-Mechanismen zum Einsatz kommen. Frameworks wie Meteor helfen dieses Problem zu entkräften, in dem sie eine klare Struktur fördern und Debug-Tools zur Verfügung stellen. Das größte Risiko in der isomorphen JavaScript-Architektur besteht darin, dass keine sicherheitsrelevanten, sensiblen Daten irrtümlich an den Client exponiert werden. Entwickler müssen deshalb mehr als bei traditionellen Frameworks darauf achten, wo der Code ausgeführt wird und wer alles darauf zugreifen kann. (Requena, 2015)

3.1.2 Latency-Compensation

In traditionellen Web-Anwendungen wartet der Benutzer bei einer Datenbankabfrage, bis der Server die Anfrage bearbeitet und an den Client zurückgesendet hat. Mittels Latency-Compensation wird diese Wartezeit umgangen und das Resultat sofort in der Benutzeroberfläche dargestellt. Es wird dadurch dem Benutzer der Eindruck einer Desktop-Anwendung vermittelt. Meteor hat dieses Prinzip von Grund auf in sein Framework eingebaut. (Susiripala, 2014b)

Die Abbildung 3.4 veranschaulicht einen typischen, traditionellen Ablauf einer Benutzerinteraktion ohne Latency-Compensation. Der Benutzer initiiert dabei einen Request an den Server via Asynchronous JavaScript and XML (AJAX). Der Server verarbeitet den Request, führt die angeforderte Methode aus und sendet das Resultat an den Client zurück. Anschließend wird der Benutzer über die Änderung informiert, indem das User Interface (UI) aktualisiert wird. Durch dieses Prinzip entsteht eine Wartezeit zwischen der Initiierung des Requests und der Auslieferung des Resultats. Oft weiß dabei der Benutzer nicht, ob seine Interaktion erfolgreich abgesetzt wurde oder nicht, wodurch die Benutzerfreundlichkeit beeinträchtigt wird. Fortschrittsanzeigen helfen dieses Problem zu umgehen, eine bessere Alternative ist jedoch das Prinzip von Latency-Compensation, wie in Abbildung 3.5 dargestellt. (Susiripala, 2014b)

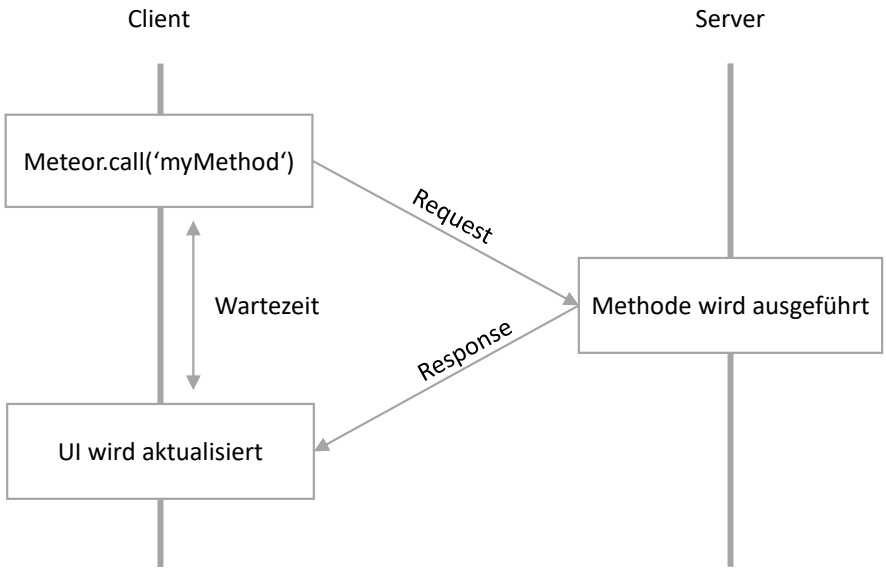

Abbildung 3.4: Ablauf einer Benutzerinteraktion ohne Latency-Compensation [Abbildung adaptiert von Susiripala (2014b)]

Mit Latency-Compensation wird ebenfalls ein Request an den Server gesendet. Der wesentliche Unterschied liegt darin, dass in diesem Fall sofort nach Aufruf der Methode das UI augenblicklich aktualisiert wird. Meteor erkennt auf dem Client die gewünschten Änderungen, simuliert das Resultat und aktualisiert ohne merkliche Verzögerung die Benutzeroberfläche. Der Server bearbeitet wie üblich die Anfrage und sendet das Resultat an den Client zurück. Meteor vergleicht nun das simulierte Resultat auf dem Client mit dem Ergebnis des Servers. Es besteht die Möglichkeit, dass serverseitig eine Anfrage abgelehnt oder ein Fehler ausgelöst wird. In einem solchen Szenario wird das Ergebnis des Servers vorrangig behandelt und der Client stellt automatisch die simulierten Änderungen zurück. Für diesen Zweck verfolgt Meteor die Modifikationen, welche während einer Simulation vorgenommen werden. Durch das Zurücksetzen eines bereits simulierten Zustandes können Benutzer irritiert werden. Je nach Verzögerung in der Datenübertragung zwischen Server und Client ist allerdings lediglich ein kurzes Flackern der Anzeige erkennbar. In den meisten Fällen kann Meteor jedoch das Ergebnis korrekt simulieren und es treten keine Unterschiede zwischen Server und Client auf. (Susiripala, 2014b)

Meteor kann aufgrund des Prinzips 'Database Everywhere', welches ebenfalls dem Framework zugrunde liegt, die clientseitige Simulation ermöglichen. Auf dem Client verwendet Meteor dazu Minimongo als lokale Datenbank. Minimongo ist ein Node-Modul und stellt eine Teilmenge der MongoDB-API, der populären NoSQL-Datenbank, zur Verfügung. In dieser Minimongo-Instanz speichert Meteor einen Teil der Serverdaten ab und kann dadurch in den meisten Fällen gute Simulationsergebnisse generieren. (Greif, 2014)

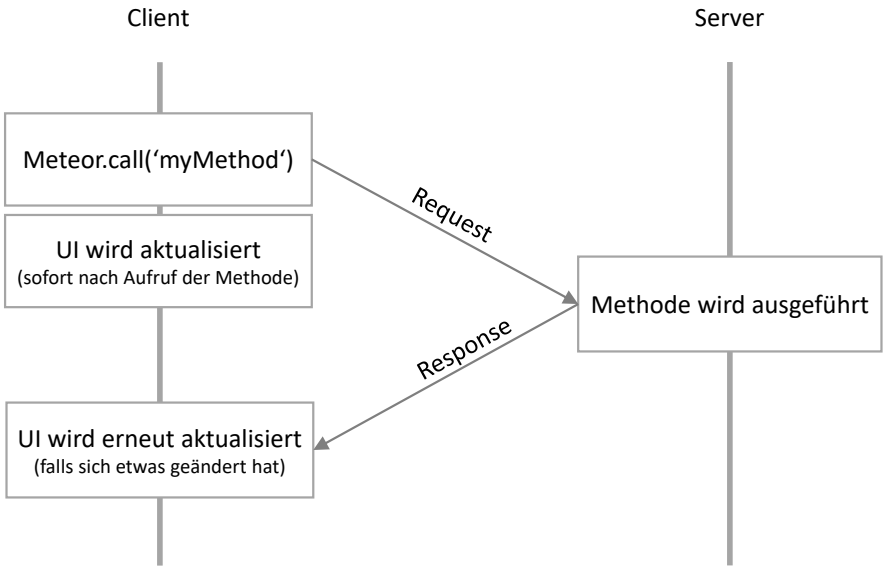

Abbildung 3.5: Ablauf einer Benutzerinteraktion mit Latency-Compensation [Abbildung adaptiert von Susiripala (2014b)]

3.1.3 Full-Stack Reactivity

Moderne Software-Anwendungen sind in hohem Maße interaktiv und werden sowohl von internen als auch externen Ereignissen gesteuert. Solche ereignisgesteuerten Applikationen reagieren auf Events, führen entsprechende Methoden aus und aktualisieren die Benutzeroberfläche und den Status der Applikation. Üblicherweise ist dabei gerade die Benutzeroberfläche der Teil der Anwendung, welcher die größte Interaktivität aufweist und typischerweise mehrere Ereignisse koordiniert und darauf reagieren muss. Aus Entwickler-Sicht sind solche Applikationen mit dem konventionellen, sequentiellen Programmierparadigma schwer zu programmieren, weil es unmöglich ist, die Reihenfolge von externen Ereignissen

vorherzusehen und zu kontrollieren. In diesem Zusammenhang wird von 'inverted control', also von der Umkehrung der Kontrolle gesprochen. Der Kontrollfluss über das Programm wird von externen Events gesteuert und ist daher nicht mehr durch den Programmierer vorgegeben. Darüber hinaus muss der Programmierer bei einer Statusänderung dafür Sorge tragen, dass alle Abhängigkeiten ebenfalls aktualisiert werden. Eine solche manuelle Verwaltung der Statusänderungen und Abhängigkeiten ist komplex und fehleranfällig. Traditionelle Programmierparadigmen setzen meist Event-Handler (Callbacks) ein. Der Nachteil von Callbacks ist, dass sie selbst für geübte Programmierer schwer beherrschbar sind, weil mehrere isolierte Codeteile die gleichen Daten bearbeiten können und dabei die Reihenfolge nicht vorhersehbar ist. Zudem besitzen Callbacks keine Rückgabewerte und müssen daher mit Seiteneffekten arbeiten, damit sie den Status der Anwendung ändern können. In der Literatur ist die Verwaltung von Callbacks unter der unrühmlichen Bezeichnung 'Callback Hell' bekannt. Aus diesem Grund ist der Bedarf nach einer Abstraktionsebene auf Programmiersprachen-Ebene gegeben, welche die Verwaltung von Events und des Status der Applikation vereinfacht. (Bainomugisha u. a., 2013)

Reaktive Programmierung ist ein Programmierparadigma, welches eine Lösung für das Problem der 'Callback Hell' verspricht. Es bewältigt die Probleme, welche von ereignisgesteuerten Anwendungen ausgehen, indem es Abstraktionen für die Formulierung von Reaktionen auf Events bereitstellt und dabei übernimmt die Programmiersprache automatisch die Verwaltung für den zeitlichen Ablauf und von jeglichen Abhängigkeiten. Ähnlich wie 'Garbage Collectors' die Speicherverwaltung abstrahieren, so abstrahieren reaktive Programmiersprachen die Verwaltung über den zeitlichen Ablauf von Ereignissen. Dieses Paradigma ist nicht neu und kann schon in Tabellenkalkulations-Systemen, wie Microsoft Excel, beobachtet werden. Formeln in Zellen einer Tabellenkalkulation werden automatisch neu berechnet, sobald sich ein Wert einer verknüpften Zelle geändert hat. Die Ursprünge der wissenschaftlichen Betrachtung der reaktiven Programmierung liegen in der funktionalen Programmiersprache Fran (Functional Reactive Animation). Dabei handelt es sich um eine Domänen-spezifische Sprache, welche in den späten 90er-Jahren entwickelt wurde und die Erstellung von Grafiken und interaktiven Medienanwendungen erleichtert. Seit diesen Anfängen wurde eine Vielzahl von Bibliotheken und Erweiterungen auch für andere Programmiersprachen entwickelt, damit sie die reaktive Programmierung unterstützen. (Bainomugisha u. a., 2013)

Die meisten Umsetzungen des reaktiven Programmierparadigma unterstützen zwei Arten der Abstraktion. Einerseits zeitveränderliche, diskrete Werte, welche auch als 'event streams' bezeichnet werden. Event streams können als asynchrone Abstraktion eines fortlaufenden Datenflusses wiederkehrender Ereignisse beschrieben werden. Tastatureingaben sind ein typisches Beispiel für event streams. Andererseits zeitlich, kontinuierliche Werte, welche als 'behaviours' oder 'signals' bezeichnet werden. Behaviours sind eine Abstraktion eines ununterbrochenen, stetigen Datenflusses von statischen Ereignissen. Zum Beispiel kann ein Zeitgeber (timer) als behaviour repräsentiert werden. (Kambona u. a., 2013)

Reaktive Programmierung ermöglicht eine deklarative Beschreibung von ereignisgesteuerten Anwendungen. Das Konzept erlaubt es Entwicklern zu formulieren, was zu tun ist und die Programmiersprache übernimmt automatisch die Verwaltung, wann es ausgeführt wird. Statusänderungen werden durch ein zugrundeliegendes Ausführungsmodel automatisch und effizient über das Netzwerk an alle Abhängigkeiten weitergeleitet. Listing 3.1 zeigt ein einfaches Beispiel mit einer Aufsummierung zweier Variablen, welche die Weiterleitung von Änderungen in reaktiven Programmiersprachen veranschaulicht. In der konventionellen, sequentiellen Programmierung erhält die Variable c den Wert 3 aus den initialen Werten der beiden anderen Variablen a und b. Die Variable c behält diesen Wert über die Zeit bei, egal ob sich die Werte der beiden anderen Variablen ändern oder nicht, außer der Wert wird wiederum explizit geändert. In der reaktiven Programmierung wird der Wert der Variable c stets aktuell gehalten, indem er automatisch neu berechnet wird, sobald sich die Werte der Variable a oder b ändern. Hinter diesem Konzept steckt die wesentliche Intention der reaktiven Programmierung. In der Terminologie der reaktiven Programmierung ist die Variable c abhängig von den Variablen a und b. Diese Abhängigkeiten können

als Graph, wie in Abbildung 3.6 illustriert, dargestellt werden. (Bainomugisha u. a., 2013)

```
a = 1
b = 2
c = a + b
```

Listing 3.1: Einfaches Beispiels zur Veranschaulichung der Weiterleitung von Änderungen in reaktiven Programmiersprachen [Code adaptiert von Bainomugisha u. a. (2013)]

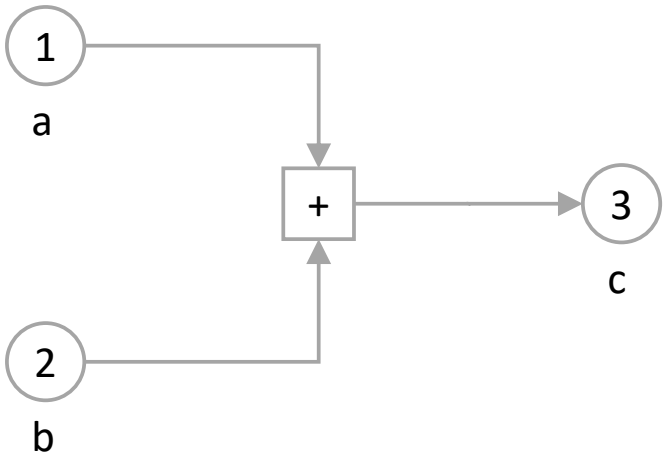

Abbildung 3.6: Abhängigkeitsgraph anhand eines einfachen Beispiels in der reaktiven Programmierung [Abbildung adaptiert von Bainomugisha u. a. (2013)]

Bei der Entwicklung des Meteor-Frameworks hatte die Unterstützung der reaktiven Programmierung von Beginn an eine hohe Bedeutung. Dabei sollte das Framework möglichst einen großen Teil abstrahieren, damit auch Einsteiger ohne Vorkenntnisse über das Paradigma einfach und schnell reaktive Anwendungen entwickeln können. Meteor verwendet dazu eine kleine, aber leistungsfähige JavaScript-Bibliothek mit dem Namen Tracker[2]. Im Wesentlichen ist Tracker eine Schnittstelle, welche es einer reaktiven Datenquelle erlaubt, mit Abonnenten der Daten zu kommunizieren, ohne die dazwischenliegende Anwendungslogik zu involvieren. Die Tracker-Bibliothek definiert dabei zwei Arten von Objekten:

- Computation - Das Computation-Objekt repräsentiert eine Funktion
- Dependency - Das Dependency-Objekt repräsentiert eine Liste an Computation-Objekten

Die wichtigste Methode der Bibliothek Tracker.autorun() erzeugt einen reaktiven Kontext, indem ein Computation-Objekt erstellt wird. Das Computation-Objekt ist dabei mit der als Parameter übergebenen Funktion verknüpft. Listing 3.2 veranschaulicht anhand eines einfachen Beispiels die Funktionsweise der Tracker-Bibliothek in Meteor. Durch Aufruf der Funktion Tracker.autorun() wird ein neues Computation-Objekt erzeugt und der globalen Tracker.current-Eigenschaft zugeordnet. Die Funktion, welche Tracker.autorun als Parameter übergeben wird, ruft Model.getValue() auf und initialisiert ein neues Dependency-Objekt. Die depend()-Funktion fügt das aktuelle Computation-Objekt zu der Liste an Computation-Objekten hinzu, welche von _valueDependency verwaltet wird. Sobald die Methode setValue() aufgerufen wird, wird die changed()-Funktion des Dependency-Objektes aufgerufen. Sie iteriert einfach durch alle Computation-Objekte innerhalb der Dependency und ruft bei jedem Objekt die invalid()-Funktion auf, welche wiederum dafür sorgt, dass alle assoziierten Funktionen mit der Computation erneut ausgeführt werden. Die erneute Ausführung erfolgt jedoch nicht sofort, sondern wird

[2]Meteor Tracker - http://docs.meteor.com/api/tracker.html

initiiert, sobald der Prozessor sich im Leerlauf befindet. Ist eine sofortige Ausführung erforderlich, kann die Funktion Tracker.flush() aufgerufen werden. (Walther, 2014)

```
Tracker.autorun(function() {
    let value = Model.getValue();

    View.setValue(value);
});

// Model
_valueDependency: new Tracker.Dependency;
getValue: function() {
    this._valueDependency.depend();
    return this._value;
}

// View
setValue: function(value) {
    if (value != this._value) {
        this._value = value;
        this._valueDependency.changed();
    }
}
```

Listing 3.2: Pseudocode zur Veranschaulichung der Funktionsweise von Tracker [Code adaptiert von Walther (2014)]

Die im vorherigen Absatz erläuterte Funktionsweise der Tracker-Bibliothek beschreibt die technische Implementierung der reaktiven Programmierung in Meteor. Üblicherweise arbeiten Entwickler in Meteor nicht direkt mit Computation- oder Dependecy-Objekten, sondern mit reaktiven Datenquellen, welche eine höhere Abstraktionsebene repräsentieren und vom Framework bereitgestellt werden. Zu diesen Datenquellen gehören zum Beispiel Session-Variablen, benutzerdefinierte reaktive Variablen, Mongo-Cursor, Meteor.user() und Meteor.status(). Alle diese Objekte erzeugen intern automatisch Tracker-Dependency-Objekte, somit ist jeder Code, welcher mit diesen Objekten interagiert, reaktiv. Meteor ermöglicht dadurch Anwendungen zu entwickeln, welche automatisch auf Datenänderungen reagieren und die entsprechenden Abhängigkeiten aktualisieren - ohne dabei viel und komplizierten Code schreiben zu müssen. (Walther, 2014)

3.2 Meteor intern verwendete Technologien

Für viele Anwender erscheint Meteor beinahe magisch, weil das Framework viele Aufgaben bei der Entwicklung von Webapplikationen erheblich erleichtert. Dahinter steckt natürlich keine Magie, sondern moderne Web-Technologien, welche in diesem Kapitel im Detail beschrieben werden.

3.2.1 Node

Meteor verwendet intern Node als Server. Node ist eine serverseitige JavaScript-Plattform, welche auf Google's JavaScript Runtime Engine V8 basiert. Bei der Entwicklung von V8 und Node wurde der Fokus auf Performance und geringer Speicherbedarf gelegt, weshalb beide Plattformen auch in C und C++ implementiert wurden. Die V8 Engine wird im Google Chrome Browser eingesetzt und ist daher für clientseitiges JavaScript im Browser konzipiert. Im Unterschied dazu unterstützt Node serverseitige Prozesse, welche nebenläufig und länger dauern können. Dabei beruht Node nicht, wie viele andere

moderne Plattformen, auf einer Multithreading-Umgebung, sondern basiert auf einem asynchronen I/O-Event-Modell-System. Singlethread-Programmierung mittels Events bietet eine effizientere und besser skalierbare Alternative zur Multithreading-Programmierung. Multithreading ist oft fehleranfällig und häufig durch Kontrollverlust gekennzeichnet, da das Betriebssystem entscheidet, welcher Thread ausgeführt wird und für wie lange. (Tilkov und Vinoski, 2010)

Node verwendet einen sogenannten Event-Loop, mithilfe dessen die Programmausführung nicht durch I/O-Aktivitäten, wie Netzwerk- oder Festplattenzugriffe blockiert wird. Durch Bereitstellen von Callback-Funktionen wird auf die Beendigung der I/O-Aktivität gewartet, während das restliche Programm weiter fortgesetzt wird. Anhand des Pseudocodes in Listing 3.3 wird versucht, die Funktionsweise des Node-Event-Loop zu erläutern. (Susiripala, 2014b)

```
// Funktionsaufrufe
fetchWeatherForecast('graz');
resizeImage('/images/largeImage.png', '/thumbnails/smallImage.png', 0.5);

// Funktionsdeklarationen
function fetchWeatherForecast(location) {
    WeatherAPI.getForecast(location, function (forecast) {
        Model.setForecast(forecast.location, forecast.temperatures,
           function () {
            console.log('weather forecast saved');
        });
    });
}

function resizeImage(source, destination, factor) {
    File.getFile(source, function (err, file) {
        var resizedImage = ImageModule.resize(file, factor);
        File.saveFile(destination, function () {
            console.log('image saved');
        });
    });
}
```

Listing 3.3: Pseudocode mit zwei unterschiedlichen Aufgaben, zur Veranschaulichung der Funktionsweise des Node-Event-Loop [Code adaptiert von Susiripala (2014b)]

Die Abbildung 3.7 zeigt schematisch, wie die beiden Funktionen über die Laufzeit ausgeführt werden. Alle Aktivitäten innerhalb fetchWeatherForecast sind dabei orange und alle Aktivitäten innerhalb resizeImage grün dargestellt. Dunkle Farben zeigen dabei die CPU-Zeit und helle Farben die Zeit für I/O-Zugriffe an. Zusätzlich veranschaulichen rote Balken die Leerlaufzeit respektive blaue Balken die Wartezeit der CPU. Bei genauerer Betrachtung der Abbildung 3.7 ist erkennbar, dass I/O-Zugriffe das Programm nicht blockieren, egal wie lange sie dauern. Beispielhaft dafür wartet die Funktion ImageModule.resize nicht bis die Funktion WeatherAPI.getForecast mit ihrer I/O-Aktivität fertig ist, bevor sie selber ausgeführt wird. Im Unterschied dazu blockieren CPU intensive Prozesse, wie die Funktion ImageModule.resize sehr wohl den Event-Loop. Das ist auch der Grund, wieso Node nicht die optimale Plattform für CPU-intensive Prozesse, wie Bild- oder Videobearbeitung ist. Ein weiterer Nachteil besteht darin, dass die intensive Verwendung von Callback-Funktionen umständlich in der Handhabung ist und die Lesbarkeit des Codes grundlegend verschlechtert wird. Die Wartbarkeit und Skalierbarkeit wird dadurch wesentlich beeinträchtigt. Zur Lösung oder zumindest zur Minderung des Problems existieren mehrere Techniken wie Promises, Generatoren, Coroutines oder Fibers. Letztere Technik wurde von den Entwicklern von Meteor für die Implementierung der Meteor-API gewählt und wird im nächsten Kapitel 3.2.2 genauer erklärt. (Susiripala, 2014b)

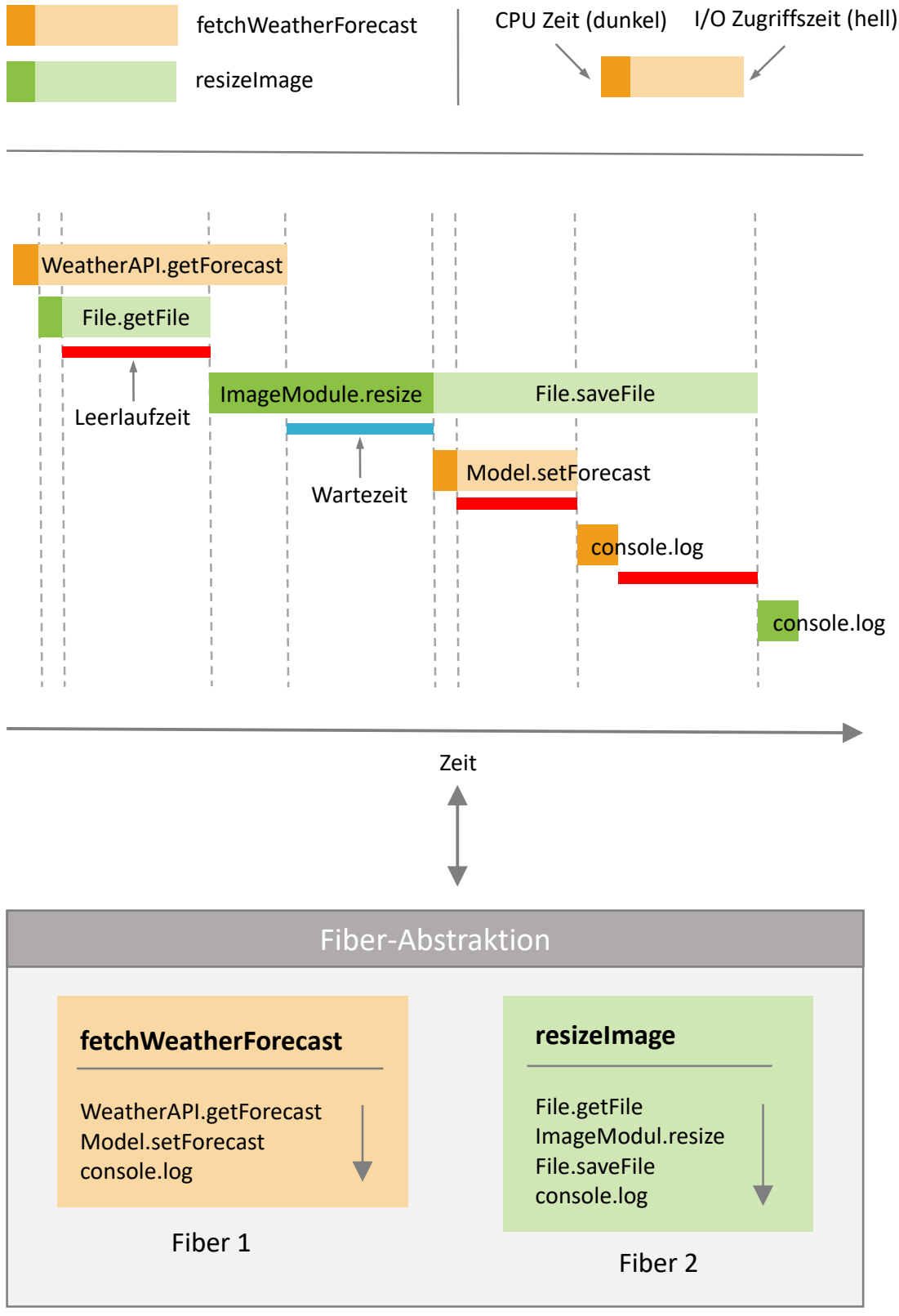

Abbildung 3.7: Schematisches Laufzeitdiagramm zur Illustrierung der Funktionsweise des Node Event-Loop und die korrespondierende Fibers-Abstraktion [Diagramm adaptiert von Susiripala (2014b)]

3.2.2 Fibers

Fibers stellen eine Abstraktion für den im Kapitel 3.2.1 beschriebenen Event-Loop bereit, welche es erlaubt, Funktionen sequentiell zu deklarieren. Abbildung 3.7 zeigt anhand des Beispiels zur Funktionsweise des Event-Loops, die Abstraktion mittels Fibers. Damit wird Entwicklern die Möglichkeit gegeben, asynchronen Code ohne Callbacks zu schreiben, ohne sich mit dem Event-Loop im Detail beschäftigen zu müssen. Diese Aufgabe übernimmt im Hintergrund Fibers und verursacht dabei keine wesentlichen zusätzlichen Kosten. Meteor fügt darüber hinaus eine Abstraktionsebene hinzu, indem es Fibers intern in seinen APIs implementiert. Dadurch benötigen Entwickler grundsätzlich auch keine wesentlichen Vorkenntnisse über Fibers und können trotzdem asynchronen Code im synchronen Stile schreiben. Meteor erstellt dazu für jeden Request vom Client automatisch einen neuen Fiber. (Susiripala, 2014b)

Coroutines sind das Fibers zugrundeliegende Konzept, auf dem die Abstraktion aufbaut. Das Konzept wurde bereits in den frühen 60er-Jahren konzeptioniert und geht auf Conway zurück, der Coroutines als 'Unterprogramme, welche als Hauptprogramme agieren' beschrieben hat. Er hat dieses Konzept für ein einfacheres Zusammenwirken zwischen den lexikalischen und syntaktischen Analyseprogrammen des COBOL-Compilers angewendet. Der Unterschied zwischen Coroutines und traditionellen Funktionen liegt darin, dass Coroutines ihren Ablauf unterbrechen und später wiederaufnehmen können. Marlin hat in seiner Dissertation 1980 die fundamentalen Eigenschaften von Coroutines folgendermaßen zusammengefasst:

- Die Werte lokaler Variablen einer Coroutine bleiben zwischen aufeinanderfolgenden, erfolgreichen Aufrufen persistent bestehen.
- Die Ausführung einer Coroutine wird unterbrochen, sobald die Kontrolle des Programms weitergegeben wird und an der gleichen Stelle fortgesetzt, wenn die Kontrolle an die Coroutine erneut übergeben wird.

(Moura und Ierusalimschy, 2009)

Das Konzept von Coroutines fand in unterschiedliche Bereiche Einzug, beispielsweise in die Simulationen, künstliche Intelligenz oder bei der Manipulation von Datenstrukturen. Bis auf wenige Ausnahmen haben jedoch Designer von Programmiersprachen dieses Konzept nicht aufgegriffen. Ein bedeutsamer Grund war, dass lange Zeit eine einheitliche, präzise Definition über Coroutines gefehlt hat, was auch dazu führte, dass unterschiedliche Implementierungen für die Unterstützung von Coroutines entwickelt wurden. Ein weiterer Grund war, dass sich Threads als de facto Standard für simultane Konstrukte durchgesetzt haben. Heute nimmt das wissenschaftliche Interesse an Coroutines speziell in zwei Szenarien wieder zu. Einerseits als Alternative zu Mutlithreading und andererseits im Kontext von Skriptsprachen, wie Python, Perl oder JavaScript. In letzterem Fall können eingeschränkte Formen von Coroutines die Implementation von einfachen Iteratoren und Generatoren unterstützen. (Moura und Ierusalimschy, 2009)

3.2.3 MongoDB

MongoDB ist eine leistungsfähige, flexible, schemafreie und skalierbare Datenbank für Daten, welche nicht transaktionsbasiert vorgehalten werden müssen. Es handelt sich dabei um keine relationale Datenbank, sondern um eine dokumentenorientierte Lösung. Der Hauptgrund, weshalb MongoDB kein relationales Modell verwendet, ist die einfachere Skalierbarkeit, aber es gibt auch andere Gründe, die dafür sprechen. Eine dokumentenorientierte Datenbank ersetzt das Konzept einer Zeile innerhalb einer Tabelle mit dem flexibleren Modell eines Dokumentes. Indem eingebettete Dokumente und Arrays unterstützt werden, liefert die dokumentenorientierte Lösung eine Möglichkeit, komplexe hierarchische Relationen in einem einzelnen Eintrag abzubilden. (Chodorow, 2013)

Eine in MongoDB erstellte Datenbank verfügt über keine vorher festgelegten Schemas. Die Schlüssel und Werte innerhalb eines Dokumentes haben weder einen fixen Datentyp noch eine festgelegte Größe. Ohne ein fixes Schema wird das Hinzufügen und Entfernen von Feldern bei Bedarf enorm vereinfacht. Generell wird die gesamte Entwicklung einer Applikation beschleunigt, weil damit häufigere Iterationszyklen ermöglicht werden. Ein weiterer Vorteil besteht darin, dass bei der Entwicklung unterschiedliche Datenmodelle ausprobiert und schlussendlich das Beste für einen jeweiligen Zweck ausgewählt werden kann. (Chodorow, 2013)

Die Datenmenge von Applikationen wächst stetig in einem unglaublich hohen Tempo. Diese Tatsache führt dazu, dass die Anforderungen an die Skalierbarkeit von Datenbanken ebenfalls steigen. Dabei kann eine Datenbank entweder vertikal (Hinzufügen leistungsfähigerer Hardware) oder horizontal (Partitionierung der Daten über mehrere Server) skaliert werden. Die vertikale Skalierung ist häufig der Weg mit dem geringsten Widerstand, sie hat jedoch erhebliche Nachteile: Leistungsfähige Hardware verursacht hohe Kosten und das Hinzufügen von neuen Kapazitäten kann nicht ewig fortgesetzt werden, weil irgendwann physikalische Grenzen erreicht werden. Die Alternative besteht in der horizontalen Skalierung, indem neue Server zu einem Cluster hinzugefügt werden. Dieser Ansatz ist billiger und erhöht die Skalierbarkeit, erschwert jedoch die Verwaltung der Infrastruktur. MongoDB wurde für eine horizontale Skalierung konstruiert. Das dokumentenorientierte Datenmodell vereinfacht die Aufspaltung der Daten auf mehrere Server. Dabei kümmert sich MongoDB automatisch um das Ausbalancieren der Daten und der Last innerhalb eines Clusters. MongoDB verteilt Dokumente automatisch und leitet Anfragen zum entsprechenden Server. (Chodorow, 2013)

MongoDB wurde als Allzweck-Datenbank konzipiert und bietet neben dem Erstellen, Lesen, Aktualisieren und Löschen von Daten auch eine ständig wachsende Anzahl an einzigartigen Features:

- Indizierung
  MongoDB unterstützt generische, sekundäre Indizes, welches eine große Anzahl an schnellen Abfragen erlaubt. Zusätzlich werden eindeutige, zusammengesetzte, räumlich-geographische und volltextfähige Indizes bereitgestellt.
- Aggregation
  MongoDB unterstützt eine Aggregationspipline, durch die komplexe Aggregationen zusammengestellt und eine Optimierung durch die Datenbank ermöglicht wird.
- Spezielle Collection
  MongoDB unterstützt sogenannte 'time-to-live'-Collections für Daten, welche nach einer bestimmten Zeit ablaufen, wie zum Beispiel Sessions. Collections mit einer fixen Größe hingegen sind nützlich für die Aufnahme von aktuellen Daten, wie zum Beispiel Protokolldaten.
- Dateispeicher
  MongoDB unterstützt ein einfach zu verwendendes Protokoll für das Speichern von großen Dateien und deren Metadaten.

(Chodorow, 2013)

Eine hohe Performance war von Beginn an ein Hauptziel bei der Entwicklung von MongoDB und viele Designentscheidungen wurden gefällt, um dieses Ziel zu erreichen. Die Kommunikation zwischen Client und Server erfolgt über ein leichtgewichtiges Protokoll und Daten werden im Hauptspeicher verwaltet. Obwohl es sich bei MongoDB um eine leistungsfähige Datenbank handelt und viele Merkmale von relationalen Datenbankmodellen übernommen wurden, wurde aufgrund der Performance bewusst auf manche Features verzichtet. (Chodorow, 2013)

3.2.4 Oplog

Bei Oplog handelt es sich um eine API von MongoDB, welche von Meteor dazu verwendet wird, auf Änderungen innerhalb der Datenbank zu reagieren. Es ist somit ein essentieller Bestandteil zur Unterstützung der sogenannten Full-Stack-Reactivity von Meteor, welche in Kapitel 3.1.3 beschrieben wird. (Susiripala, 2014b)

MongoDB verwendet Oplog primär für seine Replikations-Strategie, in der sichergestellt wird, dass Datenbanken redundant und hoch verfügbar sind. Wird Replikation in MongoDB verwendet, agiert eine Datenbank (auch als Node bezeichnet) als primäre Datenbank. Die restlichen Datenbanken - ein Replikations-Set besteht zumindest aus drei Nodes und maximal aus 12 Nodes - werden als sekundäre Datenbanken bezeichnet. Die Auswahl der primären Datenbank erfolgt komplett automatisch durch MongoDB. Fällt die aktuell aktive primäre Datenbank aus, wird eine primäre Datenbank aus den sekundären Nodes ausgewählt. Schreiboperationen können nur an der primären Datenbank durchgeführt werden, jedoch kann von jedem Node gelesen werden. Dabei ist zu beachten, dass beim Lesen von sekundären Datenbanken auch inkonsistente Daten zurückgeliefert werden können. (Susiripala, 2014b)

Oplog ist integraler Bestandteil der Replikationsstrategie von MongoDB. Es ist eine Abfolge von allen Schreiboperationen innerhalb der primären Datenbank. Technisch handelt es sich bei Oplog selbst um eine versteckte Mongo Collection mit dem Namen 'oplog.rs' innerhalb der 'local'-Datenbank. Die Collection kann mittels Abfragen auf neue oder geänderte Einträge überwacht werden. Sekundäre Datenbanken setzen exakt diese Technik zur Verwaltung der Replikation ein. Listing 3.4 zeigt beispielhaft einen Oplog Eintrag, welcher beim Hinzufügen eines neuen Eintrages in eine 'migration'-Collection der 'meteor'-Datenbank erstellt wurde. Standardmäßig benötigt Oplog 5% des verfügbaren Speicherplatzes welcher für MongoDB reserviert ist, wobei dieser Wert frei konfiguriert werden kann. (Susiripala, 2014b)

```
{
  "ts" : Timestamp(1210580223, 1),
  "h" : NumberLong("-7952051085505246239"),
  "v" : 2,
  "op" : "i",
  "ns" : "meteor.migrations",
  "o" : {
    "name" : "updatePostStatus",
    "startedAt" : ISODate("2014-06-13T16:42:55.339Z"),
    "completed" : false,
    "_id" : "HkeayNsv5kndqmLb4"
  }
}
```

Listing 3.4: Beispiel für einen Oplog Eintrag [Daten extrahiert von Susiripala (2014b)]

Meteor verwendet Oplog zur Feststellung von Änderungen in der Datenbank und informiert alle Abonnenten, welche sich für die entsprechende Änderung interessieren. Dieser Ansatz ist um ein Vielfaches effizienter als eine Lösung basierend auf Polling, also dem zyklischen Abfragen der Datenbank. In Abbildung 3.8 wird veranschaulicht, dass Meteor ähnlich agiert, wie eine sekundäre Datenbank bei MongoDB. Meteor speichert dabei die relevanten Daten von Oplog in einem internen Cache. Die Arbeit mit Oplog übernimmt gänzlich Meteor als Framework, Entwickler müssen sich deshalb meist um diesen Aspekt der Applikation nicht kümmern. (Susiripala, 2014b)

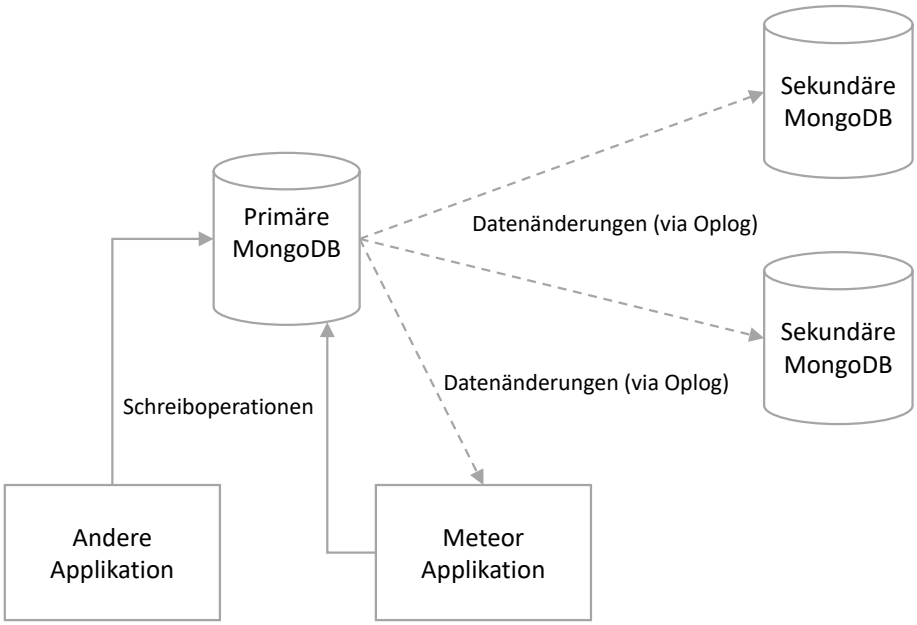

Abbildung 3.8: Verwendung von Oplog in Meteor [Diagramm adaptiert von Susiripala (2014b)]

3.2.5 DDP - Distributed-Data-Protocol

Meteor verwendet das Distributed Data Protocol (DDP) als Kommunikation zwischen Server und Client. Erst DDP in Verbindung mit Meteor ermöglicht es, wirkliche webbasierte Echtzeit-Applikationen zu erstellen, welche reaktiv auf Datenänderungen reagieren. Es handelt sich dabei um ein minimalistisches, auf JSON basiertes Protokoll. Die Implementierung in Meteor baut auf WebSockets und SockJS als Transportprotokoll. SockJS dient dazu, WebSockets zu emulieren, falls diese nicht zur Verfügung stehen. Technisch wäre auch eine Implementierung auf anderen Transportprotokollen vorstellbar. DDP-Nachrichten sind einfache JSON-Objekte mit einigen Feldern, welche als EJSON definiert sind. EJSON wird im Kapitel 3.2.6 noch genauer erläutert. Der Zweck des DDP ist einerseits die Abwicklung von Remote Procedure Call (RPC)-Aufrufen und andererseits die Datenverwaltung. RPC ermöglicht dem Client eine Methode auf dem Server aufzurufen und auf das Ergebnis zu warten. Zusätzlich bietet DDP die Möglichkeit, den Aufrufer der Servermethode zu informieren, sobald alle Schreiboperationen innerhalb der Methode auch auf alle anderen Clients reflektiert wurde. Die Datenverwaltung ist die Hauptaufgabe des DDP. Das Protokoll ermöglicht dem Client, sich bei reaktiven Datenquellen anzumelden und Benachrichtigungen über Datenänderungen zu erhalten. DDP ist eng mit MongoDB gekoppelt, jede Benachrichtigung ist daher auch an eine Collection gebunden. (Susiripala, 2014b)

Das schematische Datenflussdiagramm in Abbildung 3.9 dargestellt illustriert anhand eines Beispiels die Funktionalität des DDP. In dem konkreten Beispiel existiert für jeden Benutzer eine reaktive Datenquelle 'account', in welche andere Benutzer Transaktionen hinzufügen können. Der Benutzer Bob meldet sich zuerst bei seinem Konto (account) an und bekundend damit, dass er über alle Änderungen benachrichtigt werden möchte. In Listing 3.5 sind alle für das Beispiel relevanten DDP-Nachrichten aufgelistet:

1. Der Benutzer Bob (DDP-Client) sendet einen 'subscription'-Anfrage für sein Konto (account) an die Bank (DDP-Server).
2. Bob erhält für jede Transaktion, welche andere Benutzer (Alice und Sam) seinem Konto hinzufügen eine Benachrichtigung

3. Nachdem alle Transaktionen von der Bank versendet wurden, übermittelt das Protokoll eine 'ready'-Nachricht an Bob, welche das Ende aller initialen Daten für diese 'subscription' kennzeichnet.
4. Einige Zeit später übermittelt Richard eine Transaktion und Bob erhält wiederum eine 'added'-Nachricht vom DDP-Server.

(Susiripala, 2014b)

In ähnlicher Art und Weise kann ein DDP-Server auch 'changed'- beziehungsweise 'removed'-Nachrichten übermitteln. Die komplette DDP-Spezifikation ist Open-Source und kann online[3] eingesehen werden. (Susiripala, 2014b)

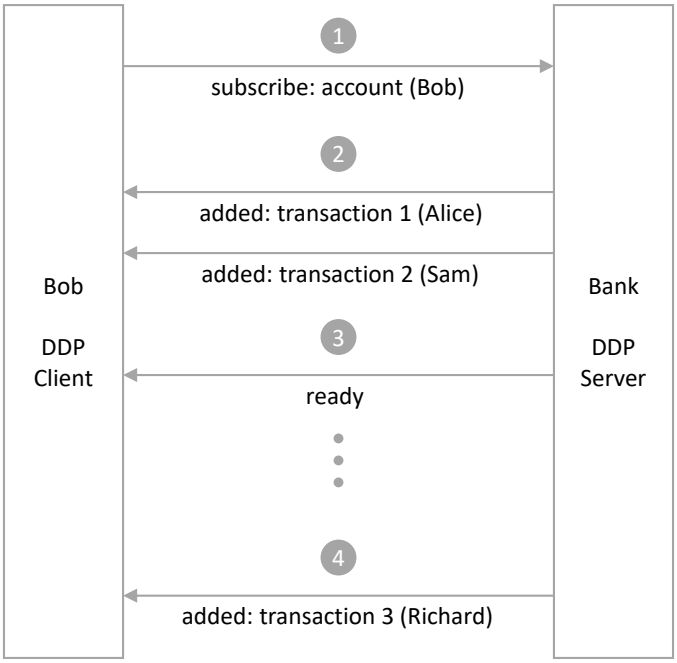

Abbildung 3.9: Beispielhaftes Datenflussdiagramm des Distributed-Data-Protocol (DDP) zur Datenverwaltung in Meteor [Diagramm adaptiert von Susiripala (2014b)]

```
1.{"msg": "sub", id: "random-id-1", "name": "account", "params": ["bob"]}
2.{"msg": "added", "collection": "transactions", "id": "record-1", "
    fields": {"amount": "20EURO", "from": "alice"}}
  {"msg": "added", "collection": "transactions", "id": "record-2", "
      fields": {"amount": "100EURO", "from": "sam"}}
3.{"msg": "ready": "subs": ["random-id-1"]}
4.{"msg": "added", "collection": "transactions", "id": "record-3", "
    fields": {"amount": "500EURO", "from": "richard"}}
```

Listing 3.5: Beispielhafter Aufbau von Distributed-Data-Protocol (DDP)-Nachrichten [Daten adaptiert von Susiripala (2014b)]

Für ein besseres Verständnis, wie Meteor intern funktioniert und für intensives Debuggen ist eine Analyse des DDP unerlässlich. Für diesen Zweck wurde das Node-Modul 'Meteor DDP Analyzer'[4] entwickelt. Dabei handelt es sich um einen einfachen DDP-Proxy, welcher DDP-Nachrichten aufzeichnet und sie visuell darstellt, wie in Abbildung 3.10 ersichtlich ist. (Susiripala, 2014b)

[3] DDP Spezifikation - https://github.com/meteor/meteor/blob/devel/packages/ddp/DDP.md
[4] Meteor DDP Analyzer - https://github.com/arunoda/meteor-ddp-analyzer

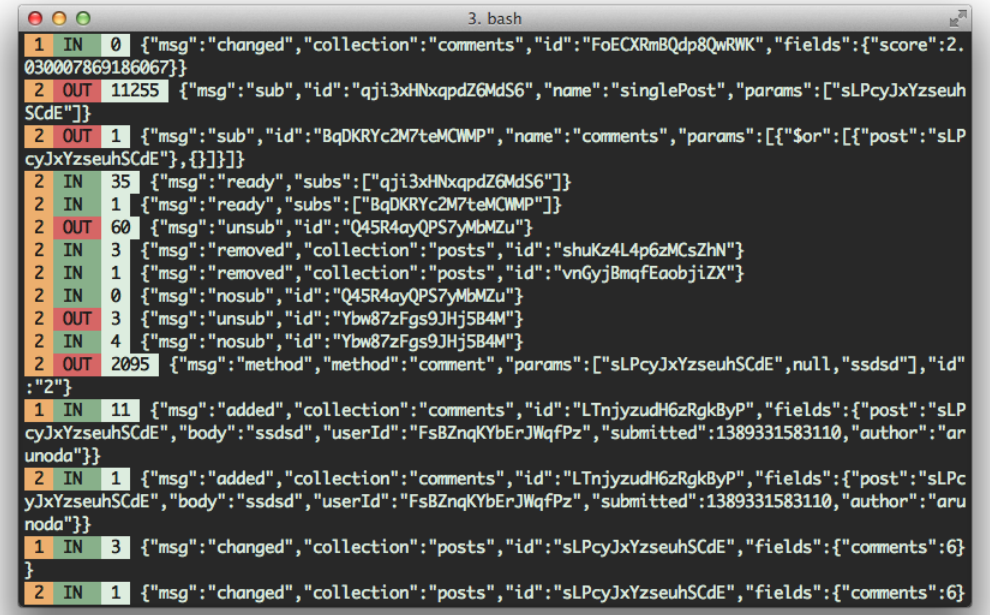

Abbildung 3.10: Screenshot des DDP-Analyzer-Tools zur Analyse des Distributed-Data-Protocol [Screenshot extrahiert von Susiripala (2014a)]

3.2.6 EJSON

EJSON ist ein Akronym für Extensible JavaScript Object Notation und wurde von Meteor entwickelt. Es unterstützt die gleichen nativen Datentypen wie JSON und zusätzlich folgende Datentypen:

- Date (JavaScript Date)
- Binary (JavaScript Uint8Array oder das Result von EJSON.newBinary)
- Benutzerdefinierte Datentypen (Mongo.ObjectID ist ein Beispiel für diesen Datentyp)

(Coleman und Nguyen, 2016)

EJSON ist eine Möglichkeit, mehr als die nativen Datentypen von JSON in das Distributed-Data-Protocol (DDP), welches in Kapitel 3.2.5 genauer beschrieben wird, einzubetten. Meteor unterstützt zusätzlich alle nativen EJSON-Datentypen als Argument und als Resultat von Server-Methoden in der MongoDB-Datenbank und in Sessions-Variablen. Werden EJSON-Objekte serialisiert, ergeben sich daraus valide JSON-Objekte. Listing 3.6 zeigt ein Beispiel eines serialisierten EJSON-Objektes mit einem Datum und einem binären Buffer als Datentypen. EJSON ist Teil der DDP-Spezifikation, ist als solche Open-Source und kann online[5] eingesehen werden. (Coleman und Nguyen, 2016)

```
{
  "d": {"$date": 1358205756553},
  "b": {"$binary": "c3VyZS4="}
}
```

Listing 3.6: Serialisiertes EJSON-Objekt mit einem Datum und einem binären Buffer [Daten extrahiert von Coleman und Nguyen (2016)]

[5]EJSON Spezifikation - https://github.com/meteor/meteor/blob/devel/packages/ddp/DDP.md#appendix-ejson

3.3 Sicherheitsaspekte in Meteor

Die Sicherheit einer Webapplikation hängt erheblich vom Verständnis und der Kenntnis möglicher Angriffsflächen zwischen unterschiedlichen Sicherheitsbereichen ab. In Meteor-Applikationen bedeutet dieser zentrale Aspekt:

- Code, welcher auf dem Server ausgeführt wird, kann vertraut werden.
- Allem andere kann prinzipiell nicht vertraut werden: zum Beispiel Client-Code oder übermittelte Daten.

(Meteor-Team, 2016)

In der Praxis bedeutet dies, dass sich der Fokus bezüglich Sicherheit auf die Schnittstellen zwischen diesen beiden Bereichen konzentrieren sollte: Einerseits die Validierung und Überprüfung jeglicher Eingabe vom Client und andererseits die Sicherstellung, dass keine geheimen und vertraulichen Informationen an den Client übermittelt werden. (Meteor-Team, 2016)

Aufgrund der Tatsache, dass Meteor-Applikationen in einem Stil geschrieben werden, indem Server- und Client-Code häufig ineinander übergehen, ist es umso wichtiger sich bewusst zu sein, welcher Teil des Codes auf dem Server und welcher Teil auf dem Client ausgeführt wird. Insbesondere in Meteor heißt das folgende Sicherheitsüberprüfungen auf jeden Fall vorzunehmen:

- Alle Daten, welche mittels Argument an Meteor-Methoden übermittelt werden, müssen validiert werden und Methoden sollten keine Daten an den Client beziehungsweise den Benutzer zurückgegeben, auf die sie keinen Zugriff haben.
- Alle Daten, welche mittels Argument an Meteor-Publications übermittelt werden, müssen validiert werden und Publications sollten keine Daten an den Client beziehungsweise den Benutzer zurückgegeben, auf die sie keinen Zugriff haben.
- Es muss sichergestellt werden, dass keine Quellcode- oder Konfigurations-Dateien mit vertraulichen Daten an den Client übermittelt werden.

(Meteor-Team, 2016)

3.4 Testen von Meteor-Applikationen

Mithilfe von Tests kann sichergestellt werden, dass die Applikation so funktioniert wie geplant, insbesondere wenn sich der Code im Laufe der Zeit ändert. Qualitativ hochwertige Tests erlauben es, Code mit einem erhöhten Vertrauen zu restrukturieren oder neu zu schreiben. Ein zusätzlicher positiver Nebeneffekt ist, dass Tests eine sehr konkrete Form der Dokumentation des erwarteten Verhaltens der Applikation darstellen. (Meteor-Team, 2016)

In den meisten Fällen unterscheidet sich das Testen einer Meteor-Applikation nicht von einer anderen JavaScript-Applikation. Im Vergleich zu herkömmlichen Frameworks, welche sich entweder auf das Backend oder das Frontend fokussieren, können zwei Aspekte das Testen in Meteor ein wenig anspruchsvoller machen:

- Client/Server-Daten
  Das Datensystem in Meteor erleichtert die Kluft zwischen Server und Client zu überbrücken und erlaubt dadurch häufig, Applikationen zu erstellen, ohne sich über den Datenaustausch Gedanken machen zu müssen. Es ist daher beim Testen entscheidend, dass der Code über diese Lücke hinweg ordnungsgemäß funktioniert. In traditionellen Frameworks wird üblicherweise viel Zeit für die Planung von Schnittstellen zwischen Server und Client investiert. Oft können diese beiden Seiten danach in Isolation getestet werden. In Meteor ermöglicht der Testmodus vergleichsweise einfach Integrationstest zu schreiben, welche den gesamten Stack abdecken. Eine weitere Herausforderung besteht darin, Testdaten im Kontext des Clients bereitzustellen.

- Reactivity
  Das Reactivity-System von Meteor ist 'abschließend konsistent', in dem Sinne, dass, sobald innerhalb des Systems sich eine reaktive Eingabe ändert, wird mit einer gewissen Verzögerung auch die Benutzeroberfläche aktualisiert. Das kann eine Herausforderung beim Testen darstellen, aber Meteor stellt einige Möglichkeiten bereit zu warten und die Ergebnisse nach Änderungen zu verifizieren.

(Meteor-Team, 2016)

Die empfohlene Vorgehensweise eine Meteor-Applikation zu testen besteht darin, den dafür vorgesehenen Befehl 'meteor test' zu verwenden. Der Befehlt setzt die Applikation in einen speziellen Testmodus. Dieser führt folgende Punkte durch:

- Ladet nicht mehr vorab den kompletten Code der Applikation, was Meteor ansonsten machen würde.
- Ladet jegliche Dateien in der Applikation, welche mit folgenden regulären Ausdrücken übereinstimmen: *.test[s].* oder *.spec[s].*
- Setzt das 'Meteor.isTest'-Flag auf 1.
- Startet das Meteor Test-Driver Paket.

(Meteor-Team, 2016)

Für die Entwicklung bedeutet diese Vorgehensweise, dass Tests in Dateien mit einem bestimmten Namen geschrieben werden können und diese Dateien nicht in das normale Produktions-Build inkludiert werden. Wenn die Applikation im Testmodus gestartet wird, werden nur diese Test-Dateien geladen, welche wiederum Module importieren können. Dieser Modus ist ideal für Unit-Tests und einfache Integrationstests. Zusätzlich bietet Meteor einen sogenannten 'Full Application'-Testmodus an, welcher mittels des Befehls 'meteor test –full-app' gestartet werden kann. Er unterscheidet sich vom herkömmlichen Testmodus in folgenden Punkten:

- Ladet Dateien, welche mit folgenden regulären Ausdrücken übereinstimmen: *.app-test[s].* oder *.app-spec[s].*
- Ladet vorab den kompletten Code der Applikation, wie es Meteor normalerweise macht.
- Setzt das 'Meteor.isAppTest' auf 1 (anstelle von 'Meteor.isTest')

(Meteor-Team, 2016)

Der 'Full Application'-Testmodus bedeutet daher, dass die komplette Applikation wie üblich geladen und ausgeführt wird. Das ermöglicht komplexere Integrationstest und Akzeptanztests zu erstellen. (Meteor-Team, 2016)

Der 'meteor test'-Befehl erwartet einen Test-Treiber als '–driver-package'-Argument. Ein Test-Treiber ist eine Mini-Applikation, welche anstelle der eigentlichen Applikation ausgeführt und dabei jeden definierten Test einzeln startet und die Ergebnisse auf eine bestimmte Art und Weise anzeigt.
Es gibt zwei grundlegende Arten von Test-Treibern:

- Web Reporters
  Dabei handelt es sich um eine Meteor-Applikation, welche eine spezielle Anzeige für die Ergebnisse von Tests innerhalb des Browsers bereitstellt
- Console Reporters
  Diese Art von Test-Treibern laufen komplett innerhalb der Konsole und werden primär für automatische Tests - zum Beispiel bei Continous Integration (CI) - angewendet.

(Meteor-Team, 2016)

3.5 Deployment von Meteor-Applikationen

Deploying, also das Bereitstellen einer Meteor-Applikation, ist vergleichbar mit jeder anderen WebSocket-basierten Node-Applikation - mit ein paar spezifischen Ausnahmen. Im Allgemeinen ist die Bereitstellung von Webapplikationen grundlegend unterschiedlich im Vergleich zu den meisten anderen Arten von Software, in dem Sinne, dass sie so oft wie gewünscht und jederzeit bereitgestellt werden können. Nichtsdestotrotz ist es dennoch wichtig, dass Änderungen innerhalb eines Qualitätssicherungsprozesses stets gründlich getestet werden, wie in Kapitel 3.4 bereits beschrieben. (Meteor-Team, 2016)

Bei der Bereitstellung von Webapplikationen ist es üblich, in drei Laufzeitumgebungen zu unterteilen:

1. Entwicklung
   Die Entwicklungsumgebung bezieht sich meist auf die lokale Umgebung, in welcher neue Features entwickelt und lokale Tests ausgeführt werden.
2. Staging
   Dabei handelt es sich um eine Zwischenumgebung, die der tatsächlichen Produktion ähnlich aber für Benutzer der Applikation nicht zugänglich ist. Staging dient in erster Linie zum Testen der Applikation und für die Qualitätssicherung.
3. Produktion
   Damit ist die tatsächliche Bereitstellung der Applikation gemeint, welche von den Benutzern verwendet wird.

(Meteor-Team, 2016)

Die Idee einer Staging-Umgebung ist das Bereitstellen einer für den Benutzer nicht zugängliche Testumgebung, welche hinsichtlich der Infrastruktur möglichst nahe der Produktionsumgebung ist. Viele häufige Fehler tauchen oft erst innerhalb der Infrastruktur des Benutzers auf, weil sie in der Entwicklungsumgebung schlicht nicht auftreten. Ein einfaches Beispiel dafür sind Fehler, welche im Zusammenhang mit der Latenz zwischen Server und Client zum Vorschein kommen. (Meteor-Team, 2016)

In Meteor existieren zwei Möglichkeiten, die Applikation außerhalb des Quellcodes zu konfigurieren:

1. Umgebungsvariablen
   Es können eine beliebige Anzahl an 'ENV_VARS' an einen laufenden Prozess gebunden werden.
2. Settings-Argument
   Dabei handelt es sich um Konfigurationen innerhalb eines JSON-Objektes, welches via dem '--settings'-Argument in der Konsole oder mittels der Umgebungsvariable 'METEOR_SETTINGS' gesetzt werden kann.

(Meteor-Team, 2016)

Die Möglichkeit mittels des Settings-Arguments sollte für Konfigurationen, welche die aktuelle Umgebung spezifisch betreffen, angewendet werden - wie es zum Beispiel Zugangs-Token zu APIs sind. Diese Konfigurationen ändern sich nicht zwischen unterschiedlichen Prozessen der Applikation innerhalb einer aktuellen Umgebung. Hingegen sollten Umgebungsvariablen für Konfigurationen, welche spezifisch einen Prozess betreffen, eingesetzt werden. Das ermöglicht eine bequeme Änderung der Konfigurationen für unterschiedliche Instanzen der Applikation. Generell gilt für alle Konfigurationen, dass sie aus Sicherheitsgründen nicht innerhalb des gleichen Repository wie der Quellcode der Applikation gespeichert werden sollten. (Meteor-Team, 2016)

Meteor ist eine Open-Source-Plattform und Meteor-Applikationen können deshalb fast überall ausgeführt werden, ähnlich wie normale Node-Applikationen. Trotzdem kann die manuelle Verwaltung der Infrastruktur, sodass Meteor-Applikationen überall wie erwartet funktionieren, aufwändig und kompliziert sein. Meteor empfiehlt daher Applikationen in der Produktionsumgebung auf Meteors eigenen Galaxy-Hosting-Plattform[6], welche speziell für solche Applikationen eingerichtet ist, zu betreiben. (Meteor-Team, 2016)

Natürlich können Meteor-Applikationen auch auf einer eigenen, benutzerdefinierten Hosting-Lösung bereitgestellt werden. Meteor stellt dafür den 'meteor build'-Befehl zur Verfügung. Der Befehl erstellt ein Deployment-Paket, welches eine reine Node-Applikation beinhaltet. Alle npm[7]-Abhängigkeiten müssen vor dem Befehl installiert werden, damit sie später im Paket inkludiert sind. Es ist darauf zu achten, das Deployment-Paket für die korrekte Architektur zu erstellen. In den meisten Fällen unterscheidet sich dabei die Architektur der Entwicklungsumgebung von jener der Produktionsumgebung. Die Architektur kann beim 'meteor build'-Befehl mittels dem 'architecture'-Parameter spezifiziert werden. (Meteor-Team, 2016)

[6] Meteor Galaxy - https://www.meteor.com/hosting
[7] npm - JavaScript-Paketmanager - https://www.npmjs.com/

Kapitel 4

Einführung in das Angular Framework

> *“ Open platforms historically undergo a lot of scrutiny, but there are a lot of advantages to having an open source platform from a security standpoint. ”*
>
> [Pichai, 2014]

Bei Angular handelt es sich um ein clientseitiges Open-Source-Framework zur Entwicklung von Single-Page-Applikation (SPA) auf Basis von HTML und JavaScript. Das Projekt wird von dem amerikanischen Unternehmen Google mitentwickelt und gewartet. Sundar Pichai, CEO von Google, hebt im eingangs erwähnten Zitat speziell die Vorteile von Open-Source-Projekten hinsichtlich der Sicherheit hervor. Über dieses Thema gibt es unter Experten unterschiedliche Meinungen. Unbestreitbar ist jedoch, dass das Thema Sicherheit innerhalb von Angular einen großen Stellenwert einnimmt und es wird deshalb auch in Kapitel 4.12 noch im Detail betrachtet.

Die erste Release-Version von Angular wurde 2010 veröffentlicht und hatte schnell an großer Popularität innerhalb der Webcommunity gewonnen. Mittlerweile liegt Angular in Version 2 vor, welche komplett neu entwickelt wurde. Aus diesem Grund besteht zwischen diesen beiden Versionen keine Rückwärtskompatibilität. Das Framework an sich besteht aus mehreren Bibliotheken, von denen einige zwingend erforderlich und einige optional sind. Als Programmiersprache können Entwickler zwischen reinem JavaScript oder Sprachen wie Dart und Typescript, welche zu JavaScript transpiliert werden, auswählen. Aufgrund der vielen nützlichen Erweiterungen von Typescript, wie Klassen, statische Typisierung, Generics oder Lambdas, empfiehlt Angular, diese von Microsoft mitentwickelte Programmiersprache zu verwenden. Typescript basiert auf den Vorschlägen zum zukünftigen ECMAScript-6-Standard und transpiliert den Code nach ECMAScript-3 oder optional auch nach ECMAScript-5. Umgekehrt ist jeder JavaScript-Code auch gleichzeitig gültiger Typescript-Code, sodass JavaScript-Bibliotheken von Drittanwendern ohne Probleme in Typescript eingebunden werden können. (Fain und Moiseev, 2016)

Abbildung 4.1 zeigt die übergeordnete Architektur mit den wesentlichen Teilen aus denen eine Angular-Applikation besteht. In den folgenden acht Unterkapiteln werden diese Elemente einzeln genauer betrachtet und im Detail erklärt.

4.1 Module

Module helfen in Angular die Applikation besser zu organisieren. Viele der Bibliotheken, aus denen Angular besteht, sind in sich Module, wie zum Beispiel das Router-Module oder das Http-Module. Genauso werden viele Drittanwender-Bibliotheken als Angular-Module zur Verfügung gestellt. Angular-Module fassen logisch zusammengehörende Komponenten (4.2), Direktiven (4.5.7) und Services (4.7)

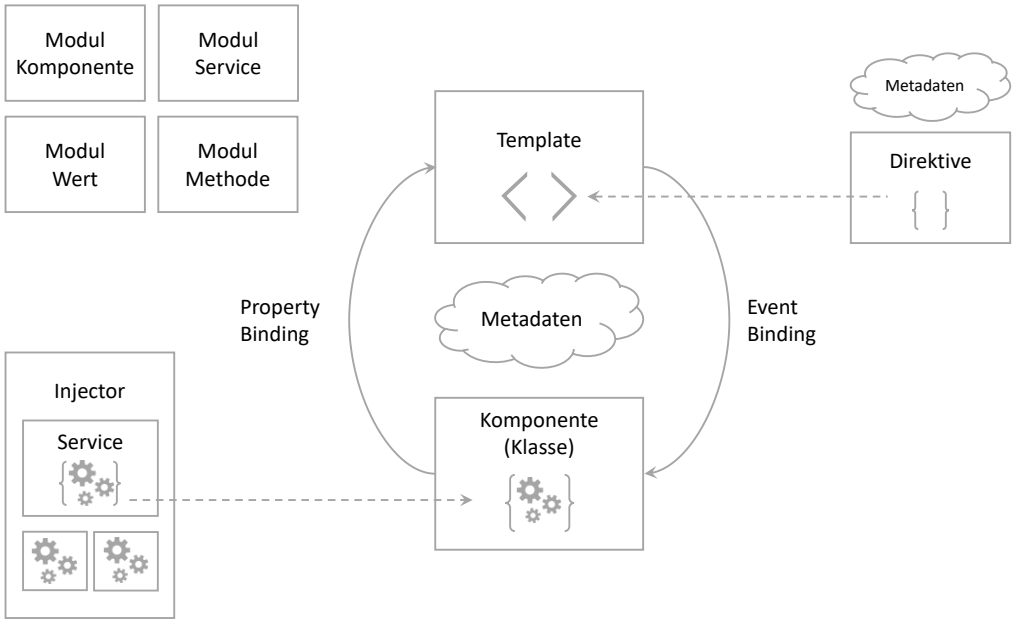

Abbildung 4.1: Angular Architektur [Abbildung adaptiert von Angular-Team (2016)]

zusammen. Dabei ist nicht strikt vorgeben, nach welchem Schema diese Modularisierung vorgenommen werden muss. Module können sich auf Features, Geschäftsbereiche oder etwa Arbeitsabläufe einer Applikation fokussieren. (Angular-Team, 2016)

Zur Performanceverbesserung können Module 'eagerly loaded', das heißt bereits während des Starts der Applikation, oder 'lazy loaded', also asynchron vom Router geladen werden. Ein Angular Modul ist gekennzeichnet durch die @NgModule-Decorator-Funktion. Decorators sind Funktionen, welche JavaScript-Klassen modifizieren. Angular verfügt über viele Decorator-Funktionen, damit das Framework Kenntnis darüber erlangt, was die jeweiligen Klassen bedeuten und welche Funktion sie ausfüllen. Dem @NgModule-Decorator wird ein Metadaten-Objekt übergeben, welches Angular mitteilt, wie das entsprechende Modul kompiliert und der enthaltene Code ausgeführt werden soll. (Angular-Team, 2016)

Jede Angular-Applikation verfügt über zumindest ein Modul, dem Root-Modul. Per Konvention ist dabei die Klasse 'AppModule' und die dazugehörige Datei 'app.module.ts' zu benennen. Das Root-Modul wird zu Beginn von Angular geladen und damit die Applikation gestartet. Listing 4.1 zeigt die einfachste Variante eines Angular-Root-Moduls. Während kleine, einfache Applikationen nur aus dem Root-Module bestehen können, haben die meisten Applikationen mehrere Feature-Module, welche logisch eine Einheit bilden. Wie bereits erwähnt definiert der @NgModule-Decorator die Metadaten des Moduls. (Angular-Team, 2016)

Die wichtigsten Eigenschaften dabei sind:

- declarations
  Eine Liste an View-Klassen (Komponenten, Direktiven oder Pipes), welche zu diesem Modul gehören.

- exports
  Eine Untermenge der in der declarations-Liste angegebenen Klassen, welche in den Templates anderer Module benötigt werden.

- imports
  Eine Liste an externen Modulen, deren exportierten Klassen in den Templates des aktuellen Moduls benötigt werden.

- providers
  Services dieses Moduls, welche zur globalen Liste an Services hinzugefügt werden. Diese Services sind anschließend in allen Teilen der Applikation verfügbar.

- bootstrap
  Die Root-Komponente, welche alle anderen Komponenten beinhaltet. Nur das Root-Modul sollte diese Eigenschaft setzen.

(Angular-Team, 2016)

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Listing 4.1: Einfachste Variante eines Angular-Root-Moduls [Code adaptiert von Angular-Team (2016)]

Wie in Listing 4.1 ersichtlich, wird zuerst ein einzelnes Hilfsmodul namens BrowserModule importiert. BrowserModule muss zwingend von jeder Applikation, welche im Browser geladen werden soll, hinzugefügt werden. Es registriert für die Applikation kritische Service-Provider und enthält allgemeine Direktiven wie NgIf oder NgFor, welche danach in allen Komponenten des Moduls zur Verfügung stehen. Listing 4.2 zeigt, wie eine Applikation durch bootstrappen des Root-Moduls gestartet wird. (Angular-Team, 2016)

```
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Listing 4.2: Bootstrapping des Root-Moduls [Code adaptiert von Angular-Team (2016)]

JavaScript verfügt ebenfalls über ein Modulsystem zur Verwaltung und Organisation von JavaScript-Objekten. Es unterscheidet sich grundlegend vom Angular-Modulsystem und sollte daher nicht miteinander verwechselt werden. Beide ergänzen sich und können nebeneinander in der Entwicklung der Applikation eingesetzt werden. In JavaScript definiert jede Datei ein Modul und alle JavaScript-Objekte, welche innerhalb dieser Datei definiert werden, gehören zu diesem Modul. Objekte können dabei als öffentlich markiert werden, indem das export-Schlüsselwort verwendet wird. Mittels dem import-Schlüsselwort können wiederum öffentlich verfügbar gemachte JavaScript-Objekte von anderen Modulen verwendet werden. (Angular-Team, 2016)

4.2 Komponenten

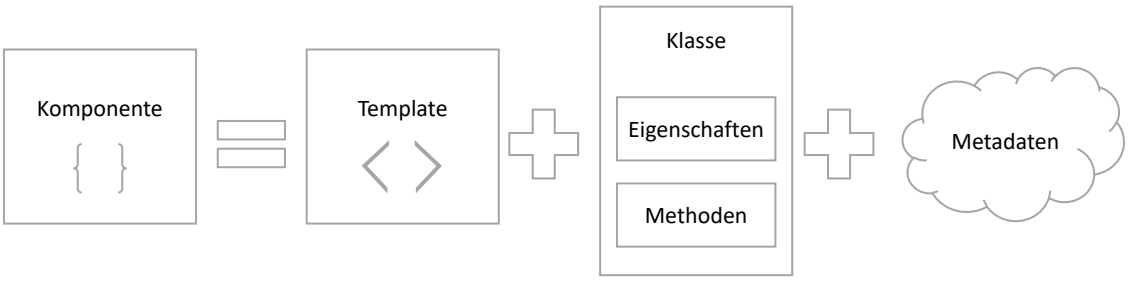

Abbildung 4.2: Aufbau einer Angular-Komponente [Abbildung adaptiert von Angular-Team (2016)]

Eine Angular-Applikation wird modular aus Komponenten aufgebaut. Jede Komponente besteht, wie in Abbildung 4.2 schematisch dargestellt, aus einem Template, einer Komponenten-Klasse, in welcher die Logik der Komponente definiert wird und aus Metadaten. Die Interaktion zwischen der Klasse und der View erfolgt dabei über eine API von Eigenschaften und Methoden. Listing 4.3 zeigt ein Beispiel einer einfachen Komponenten-Klasse namens 'ItemListComponent'. Die 'items' Eigenschaft gibt eine Liste an Item-Objekten zurück, welche von einem Service stammen. Die 'selectItem'-Methode setzt die 'selectedItem'-Eigenschaft und wird aufgerufen, sobald ein Benutzer ein Item ausgewählt hat. Angular erstellt, aktualisiert und zerstört einzelne Komponenten automatisch, wenn Benutzer durch die Applikation navigieren. Die Applikation kann auf diese Ereignisse, sogenannte Lifecycle-Hooks reagieren und Aktionen durchführen. Im vorigen Beispiel wurde auf den 'onInit'-Hook reagiert und die Items mithilfe des Service geladen. Im Kapitel 4.9 werden Lifecycle-Hooks später genauer betrachtet. (Angular-Team, 2016)

```
export class ItemListComponent implements OnInit {
  items: Item[];
  selectedItem: Item;

  constructor(private service: MyService) { }

  ngOnInit() {
    this.items = this.service.getItems();
  }

  selectItem(item: Item) { this.selectedItem = item; }
}
```

Listing 4.3: Beispiel einer Angular Komponenten-Klasse [Code adaptiert von Angular-Team (2016)]

4.3 Templates

Das Template ist jener Teil einer Angular-Komponente, welcher die visuelle Darstellung der Komponente vorgibt. Templates bestehen aus typischen HTML-Elementen und zusätzlicher Angular-Template-Syntax. In Listing 4.4 werden reguläre HTML-Elemente, wie <h1> oder <p> verwendet. Dagegen handelt es sich bei den Elementen *ngFor, (click), {{item.name}}, [item] oder <item-detail> um spezielle

Angular-Template-Syntax. Hervorzuheben im Zusammenhang mit Komponenten ist das letzte Element <item-detail>. Dabei handelt es sich um eine weitere Komponente, deren Aufgabe darin besteht, weitere Details des gerade ausgewählten Elements in der Liste darzustellen. Komponenten bilden innerhalb einer Applikation eine hierarchische Struktur, wie in Abbildung 4.3 erkennbar ist. Die 'ItemDetail'-Komponente ist eine Kind-Komponente der 'ItemList'-Komponente. (Angular-Team, 2016)

```
<h1>Meine Liste</h1>
<p>Wähle ein Element aus</p>
<ul>
  <li *ngFor="let item of items" (click)="selectItem(item)">
    {{item.name}}
  </li>
</ul>
<item-detail *ngIf="selectedItem" [item]="selectedItem"></item-detail>
```

Listing 4.4: Beispiel eines Angular-Komponenten-Templates [Code adaptiert von Angular-Team (2016)]

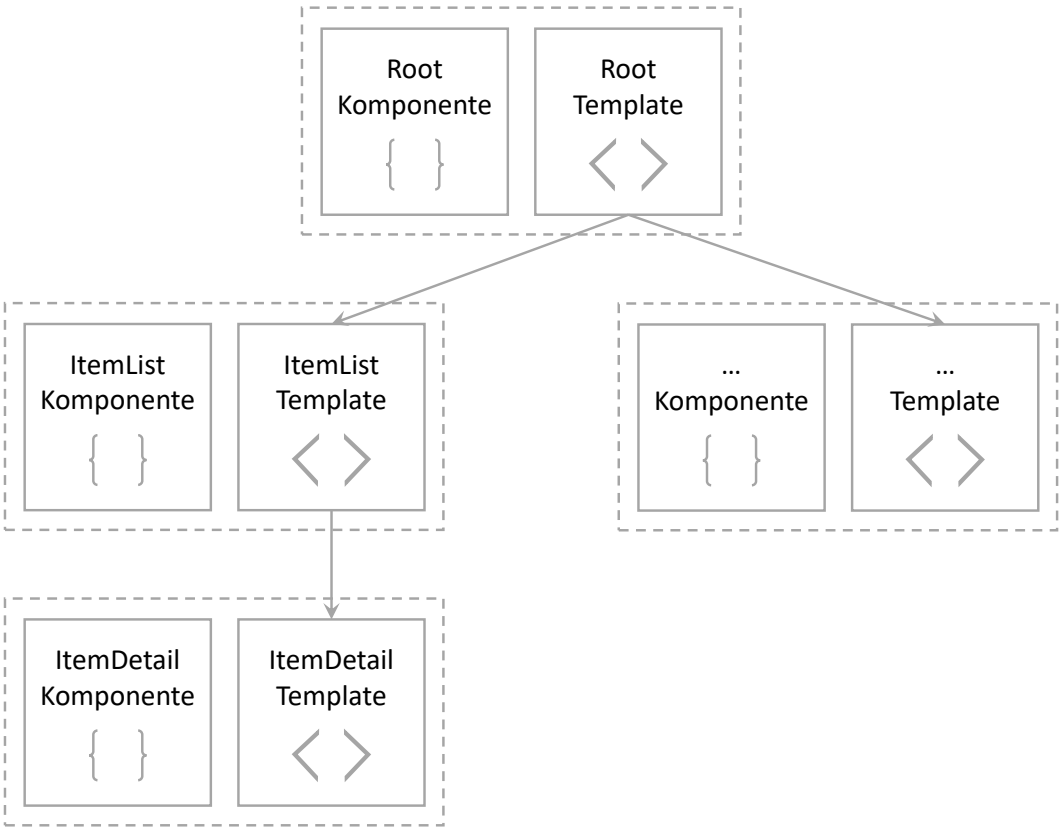

Abbildung 4.3: Angular-Template-Hierarchie [Abbildung adaptiert von Angular-Team (2016)]

4.4 Metadaten

Wird der Code in Listing 4.3 betrachtet, so handelt es sich um eine gewöhnliche Typescript-Klasse ohne jeglichen Bezug auf Angular. Damit das Framework weiß, dass es sich dabei dezidiert um eine Angular-Komponente handelt, müssen Metadaten der Klasse angehängt werden. In Typescript können Metadaten mittels einem Decorator hinzugefügt werden. Listing 4.5 zeigt beispielhaft, wie ein solcher Decorator aufgebaut ist. Der Name '@Component' indiziert dabei, dass es sich um eine Angular-Komponente handelt. Beim @Component-Decorator muss zwingend ein Konfigurations-Objekt angegeben werden. Die möglichen Konfiguration-Optionen sind an dieser Stelle nicht abschließend aufgelistet:

- moduleId
  Die moduleId gibt die Quelle der Basisadresse von relativen URL-Pfaden, wie zum Beispiel der templateUrl, an.
- selector
  Der CSS-Selektor dient zur Identifikation der Komponente innerhalb eines Template. Er weist Angular an, eine Instanz dieser Komponente zu erstellen und an der entsprechenden Stelle im Eltern-Template einzufügen. Im Beispiel 4.4 wird zwischen den Tags <item-detail> und </item-detail> die Kind-Komponente 'ItemDetail' eingefügt.
- template
  Einfache Templates können inline, das heißt innerhalb der gleichen Datei an dieser Stelle, definiert werden.
- templateUrl
  Üblicherweise werden Templates in einer separaten Datei definiert, deren relativen Pfad hier angegeben wird.
- styles / styleUrls
  Stylesheets, welche auf die Komponente angewendet werden, können ebenfalls inline oder mittels relativen Pfaden angeben werden.
- providers
  Eine Liste an Service-Providern, welche die Komponente benötigt.

(Angular-Team, 2016)

Die Metadaten sind ein wichtiger Bestandteil einer Angular Komponente, da sie Angular mitteilen, um welche Art von Code es sich handelt, wo die einzelnen Teile (Klasse und Template) zu finden sind und welche anderen Services benötigt werden. (Angular-Team, 2016)

```
@Component({
  moduleId: module.id,
  selector:    'item-list',
  templateUrl: 'item-list.component.html',
  providers:  [ MyService ]
})
export class ItemListComponent implements OnInit {
/* . . . */
}
```

Listing 4.5: Beispiel eines Decorators für eine Angular-Komponenten-Klasse [Code adaptiert von Angular-Team (2016)]

4.5 Datenbindung

Angular hilft mittels Datenbindung, Datenwerte an HTML-Elemente zu übergeben und auf Benutzerinteraktionen zu reagieren. Die Datenbindung spielt eine wichtige Rolle in der Kommunikation zwischen Komponenten, wie in Abbildung 4.4 schematisch dargestellt wird und sie koordiniert und synchronisiert Elemente des Template mit Teilen der Komponenten-Klasse. Ohne Datenbindung wäre diese Aufgabe aufwendig und eine häufige Fehlerquelle. In Tabelle 4.1 werden alle möglichen Arten der Datenbindungen von Angular gruppiert nach der Richtung des Datenflusses aufgelistet. (Angular-Team, 2016)

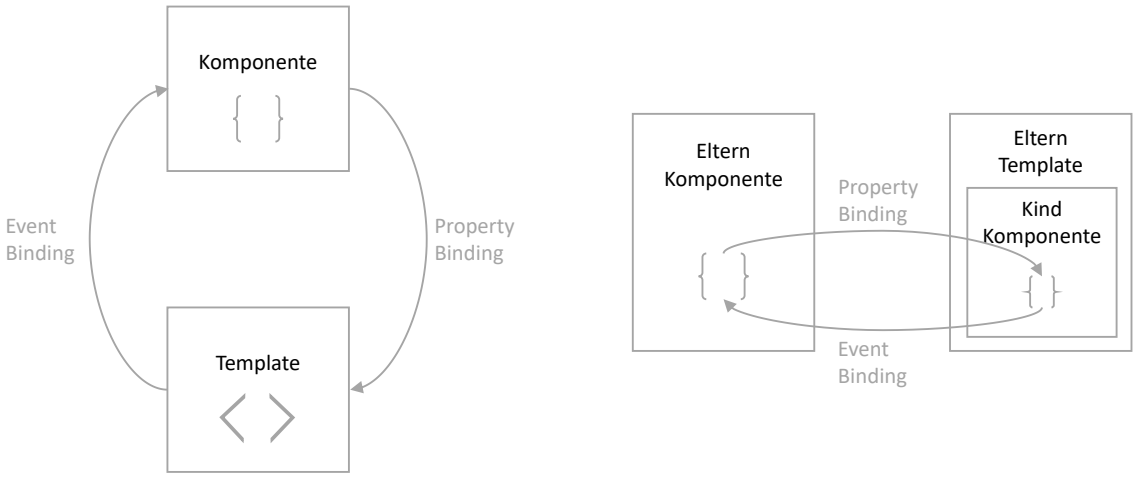

Abbildung 4.4: Datenbindung als Kommunikation zwischen Komponenten [Abbildung adaptiert von Angular-Team (2016)]

Datenfluss-Richtung	Syntax	Type
In eine Richtung Von der Datenquelle innerhalb der Komponente zum Document Object Model (DOM)	{{expression}} [target]="expression"	Interpolation Property Attribute Class Style
In eine Richtung Vom DOM zur Datenquelle innerhalb der Komponente	(target)="statement"	Event
In beide Richtungen	[(target)] = "expression"	Two-way

Tabelle 4.1: Auflistung der verschiedenen Arten der Datenbindung in Angular, gruppiert nach der Richtung des Datenflusses [Tabelle adaptiert von Angular-Team (2016)]

4.5.1 Unterscheidung zwischen HTML-Attributen und DOM-Eigenschaften

Für das Verständnis, wie die Datenbindung in Angular funktioniert, ist die Kenntnis über den Unterschied zwischen HTML-Attributen und DOM-Eigenschaften entscheidend. Grundsätzlich gilt, dass Attribute innerhalb des HTML und Eigenschaften durch das DOM definiert sind. Dabei können folgende Beobachtungen gemacht werden:

- Einige wenige HTML-Attribute haben eine 1:1-Zuordnung zu den Eigenschaften, zum Beispiel das 'id'-Attribut.
- Einige HTML-Attribute haben keine korrespondierenden Eigenschaften, zum Beispiel das 'colspan'-Attribut.
- Einige DOM-Eigenschaften haben keine korrespondierenden Attribute, zum Beispiel die 'textContent'-Eigenschaft.
- Viele HTML-Attribute erwecken den Eindruck, dass sie über eine Zuordnung zur entsprechenden DOM-Eigenschaft verfügen. Dabei initialisieren sie lediglich die Eigenschaft. DOM-Eigenschaften können sich im Unterschied zu Attribut-Werten ändern.

(Angular-Team, 2016)

Es ist wichtig festzuhalten, dass Attribute und DOM-Eigenschaften nicht dasselbe sind, auch wenn sie den gleichen Namen haben. (Angular-Team, 2016)

4.5.2 Interpolation

Interpolation ist die einfachste Möglichkeit, eine Eigenschaft der Komponenten-Klasse im Browser anzuzeigen. Dazu muss lediglich der Name der Eigenschaft innerhalb zweier geschwungener Klammern eingeschlossen werden. Angular holt sich automatisch den Wert der Eigenschaft und fügt ihn in den DOM des Browsers ein. Die Anzeige wird dabei automatisch aktualisiert, sobald sich die Eigenschaft ändert. Ganz allgemein handelt es sich beim Inhalt zwischen den beiden geschwungenen Klammern um eine sogenannte 'Template-Expression' (siehe Kapitel 4.5.3), welche Angular zuerst evaluiert und danach in einen String konvertiert. Listing 4.6 illustriert das Prinzip der Interpolation in Angular, indem zwei Zahlen innerhalb der geschwungenen Klammern addiert werden. Interpolation ist eine spezielle Syntax, welche Angular intern in ein Property-Binding konvertiert, welche im Kapitel 4.5.4 genauer erläutert wird. (Angular-Team, 2016)

```
<p>Die Summe von 1 + 2 ergibt {{1 + 2}}</p>
```

Listing 4.6: Beispiel für die Datenbindung mittels Interpolation in Angular [Code adaptiert von Angular-Team (2016)]

4.5.3 Template-Expression

Eine Template-Expression produziert einen Wert. Angular führt die Expression aus und weist den Wert dem Ziel der Datenbindung zu. Dabei kann es sich um ein HTML-Element, eine Komponente oder eine Direktive handeln. Template-Expressions können in einer JavaScript-ähnlichen Sprache verfasst werden. Dabei sind die meisten JavaScript-Anweisungen und Operatoren erlaubt, ausgenommen sind solche, die zu möglichen Seiteneffekten führen, wie zum Beispiel Zuweisungen (=, +=, -=, ...), der 'new'-Operator, Verkettungs-Operator (;) oder der Inkrement- beziehungsweise Dekrement-Operator (++ und - -). Zusätzlich werden innerhalb von Template-Expressions keine bitweisen Operatoren (| oder &) unterstützt. (Angular-Team, 2016)

Template-Expressions können die Funktionsfähigkeit und die Leistung einer Applikation stark beeinflussen. Es sollten daher folgende rudimentären Regeln - wenn möglich - eingehalten werden:

- Keine sichtbaren Seiteneffekte
  Eine Template-Expression sollte keinen Zustand der Applikation verändern, einzige Ausnahme ist der Wert des Ziels der Datenbindung.

- Schnelle Ausführung
  Angular führt Template-Expressions nach jedem 'change detection cycle' - also nach jeder Überprüfung auf Änderungen - durch. Dieser Vorgang wird durch viele asynchrone Ereignisse ausgelöst, wie zum Beispiel nach Timer-, Tastatur- oder Maus-Ereignissen. Expressions sollten daher möglichst schnell ausgeführt werden, ansonsten kann es speziell auf langsameren Geräten zu einer starken Verschlechterung der Benutzererfahrung führen.

- Fokus auf Einfachheit
  Obwohl auch sehr komplexe Template-Expressions erstellt werden können, sollte dies vermieden werden. Die Applikationslogik gehört, wenn möglich in die Komponente, in der sie einfacher entwickelt und getestet werden kann.

- Idempotenz
  Eine idempotente Expression ist ideal, weil sie frei von Seiteneffekten ist und die Leistung des Vorganges zur Überprüfung von Änderungen erhöht. In Angular bedeutet eine idempotente Expression, dass sie immer das gleiche Resultat zurück liefert, solange bis sich ein abhängiger Wert ändert. Wenn eine idempotente Expression einen String oder einen numerischen Wert zurück liefert, dann wird der gleiche Wert zurück geliefert, wenn sie zweimal oder öfter hintereinander aufgerufen wird. Der gleiche Aspekt gilt im gleichen Ausmaß auch für Objekt-Referenzen.

(Angular-Team, 2016)

4.5.4 Property-Binding

Property-Binding wird oft auch als One-Way Binding bezeichnet, weil der Wert in eine Richtung fließt - von der Eigenschaft einer Komponente in die Eigenschaft eines Ziel-Elements. Es ist daher nicht möglich, den Wert eines Ziel-Elementes auszulesen, es ist lediglich möglich, den Wert zu setzen. Gleichermaßen ist es nicht möglich, eine Methode des Ziel-Elementes mittels Property-Binding aufzurufen. Das Ziel der Datenbindung wird deklariert, indem die Element-Eigenschaft in umschließende rechteckige Klammern gesetzt wird. Zum Beispiel ist die Ziel-Eigenschaft in Listing 4.7 die 'src'-Eigenschaft des entsprechenden Bild-Elementes. (Angular-Team, 2016)

```
<img [src]="myUrl">
```

Listing 4.7: Beispiel für die Datenbindung mittels Property-Binding in Angular [Code adaptiert von Angular-Team (2016)]

Der Name des Ziels der Datenbindung ist in den meisten Fällen eine Element-Eigenschaft. Zuerst überprüft jedoch Angular, ob es sich bei dem Namen um eine Eigenschaft einer bekannten Direktive handelt. Sind beide Überprüfungen fehlgeschlagen, meldet Angular einen 'unknown directive'-Fehler. (Angular-Team, 2016)

Oft besteht die Möglichkeit, zwischen Interpolation und Property-Binding auszuwählen. Zum Beispiel liefern die Paare an Datenbindungen in Listing 4.8 das gleiche Ergebnis. Interpolation ist in vielen Situationen eine bequeme Alternative zu Property-Binding. Bei String-Werten gibt es keine technischen Gründe, eine Version zu bevorzugen. Wenn Element-Eigenschaften zu Werten, welche nicht vom Typ String sind, gebunden werden, muss zwingend die Property-Binding Methode gewählt werden. (Angular-Team, 2016)

```
<p><img src="{{myUrl}}"> interpolierte Bild</p>
<p><img [src]="myUrl"> mittels 'property binding' gebundene Bild</p>

<p><span>"{{title}}" interpolierte Titel</span></p>
<p>"<span [innerHTML]="title"></span>" mittels 'property binding'
   gebundene Titel</p>
```

Listing 4.8: Unterschied zwischen Interpolation und Property-Binding in Angular [Code adaptiert von Angular-Team (2016)]

4.5.5 Event-Binding

Event-Binding ist notwendig, um auf Benutzerinteraktionen wie Tastatur-, Maus- oder Touch-Ereignisse reagieren zu können. Der Datenfluss folgt bei Event-Binding im Unterschied zu Property-Binding in die entgegengesetzte Richtung: Vom Element zur Komponente. Die Syntax des Event-Binding besteht aus dem Ziel-Event, innerhalb von runden Klammern, gefolgt von einem Gleichheitszeichen und einem in Anführungszeichen gesetzten Template-Statement. Listing 4.9 illustriert beispielhaft, wie mittels Event-Binding auf ein Klick-Event reagiert und die 'onSave()'-Methode in der entsprechenden Komponente aufgerufen werden kann. Ähnlich wie beim Property-Binding überprüft Angular zuerst, ob es sich beim angegebenen Ziel der Datenbindung um eine Eigenschaft einer bekannten Direktive handelt. Beim Event-Binding richtet Angular automatisch einen Event-Handler für das Ziel-Event ein. Sobald das Ereignis ausgelöst wurde, führt der Event-Handler das deklarierte Template-Statement aus. Das Template-Statement enthält üblicherweise einen Empfänger, der eine Aktion in Reaktion auf das Ereignis ausführt, wie beispielsweise das Speichern eines Wertes vom HTML-Element in ein Datenmodel. Im Unterschied zu Template-Expressions sind Seiteneffekte in Template-Statements üblich und werden sogar erwartet. (Angular-Team, 2016)

```
<button (click)="onSave()">Sichern</button>
```

Listing 4.9: Beispiel für die Datenbindung mittels Event-Binding in Angular [Code adaptiert von Angular-Team (2016)]

4.5.6 Two-Way-Binding

Oft ist gewünscht eine Eigenschaft der Komponenten-Klasse anzuzeigen und gleichzeitig die Eigenschaft zu aktualisieren, wenn der Benutzer Änderungen vornimmt. Angular stellt für diesen Zweck eine spezielle Two-Way-Binding Syntax zur Verfügung: [(x)]. Die Syntax kombiniert die Syntax des Property-Binding mit den eckigen Klammern [x] mit der Syntax des Event-Binding mit den runden Klammern (x). Listing 4.10 illustriert beispielhaft, wie mittels der Angular Direktive 'ngModel' das Two-Way-Binding angewendet werden kann. Dabei fließt der Wert der Eigenschaft 'name' in das 'input'-Element analog zum Property-Binding. Gleichzeitig fließen Änderungen eines Benutzers, mittels Eingabe eines Wertes in das Eingabefeld, zurück in die Komponente - analog zum Event-Binding. Dabei handelt es sich technisch um eine syntaktische Erweiterung, welche offenkundig eine erhebliche Erleichterung bereitstellt, verglichen mit der separaten Anwendung von Property-Binding und Event-Binding. (Angular-Team, 2016)

```
<input [(ngModel)]="name">
```

Listing 4.10: Beispiel für die Datenbindung mittels Two-Way-Binding in Angular [Code adaptiert von Angular-Team (2016)]

4.5.7 Direktiven

In Angular existieren drei verschiedene Arten von Direktiven:

1. Komponenten-Direktiven:
   Direktiven mit einem eigenen Template, Angular-Komponenten fallen in diese Kategorie und werden im Kapitel 4.2 detailliert betrachtet

2. Struktur-Direktiven
   Ändern die Struktur des DOM, indem DOM-Elemente hinzugefügt oder entfernt werden

3. Attribut-Direktiven
   Ändern das Aussehen oder das Verhalten eines Elements

(Angular-Team, 2016)

Während des Rendering-Prozesses der Angular-Templates wird das DOM entsprechend den Anweisungen der Direktiven transformiert. Technisch betrachtet sind Angular-Komponenten Direktiven. Aufgrund der zentralen Bedeutung von Komponenten innerhalb des Angular-Frameworks werden sie aber oft separat betrachtet. Komponenten-Direktiven sind die am häufigsten verwendeten Direktiven und werden im Kapitel 4.2 genauer beschrieben. Typische Struktur-Direktiven sind *ngFor oder *ngIf. Damit können grundlegende Kontrollstrukturen innerhalb von Templates abgebildet werden. Listing 4.11 zeigt ein Beispiel für den Einsatz von Struktur-Direktiven. Dabei teilt *ngFor Angular mit, für jedes Item in der Liste von Items ein -Element innerhalb des DOM zu erstellen. Durch die *ngIf-Anweisung wird die 'ItemDetail'-Komponente nur inkludiert, wenn ein selektiertes Element (selectedItem) existiert. (Angular-Team, 2016)

```
<li *ngFor="let item of items"></li>
<item-detail *ngIf="selectedItem"></item-detail>
```

Listing 4.11: Beispiel für Struktur-Direktiven in Angular [Code adaptiert von Angular-Team (2016)]

Mittels Attribut-Direktiven können das Aussehen und Verhalten von existierenden Elementen geändert werden. In der Syntax innerhalb von Templates unterscheiden sie sich dabei nicht von üblichen HTML-Attributen. ngModel ist eine häufig verwendete Attribute-Direktive, mit deren Hilfe die beidseitige Datenbindung, wie in Kapitel 4.5 beschrieben, angewendet werden kann. Die ngModel-Direktive ändert das Verhalten eines Elements, zum Beispiel eines Input-Elements, durch setzen der 'display-value'-Eigenschaft und durch reagieren auf 'change'-Events. Neben den bereits vorgegebenen Direktiven durch Angular können auch eigene, benutzerdefinierte Direktiven erstellt werden. (Angular-Team, 2016)

4.6 Pipes

Häufig müssen Daten, bevor sie dargestellt werden, in eine benutzerfreundliche Version transformiert werden. Die Transformationen sind oft wiederkehrend und können dabei ähnlich wie Styles betrachtet werden. Angular bietet mit Pipes die Möglichkeit, solche Transformationen von Anzeigewerten zu definieren und sie anschließend deklarativ in Angular-Templates anzuwenden. Das Framework verfügt über bereits integrierte Pipes wie zum Beispiel DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe oder PercentPipe, welche sofort für Standard-Transformationen angewendet werden können. Listing 4.12 veranschaulicht beispielhaft die Verwendung der DatePipe. Der Wert der 'created'-Eigenschaft wird dabei mittels dem Pipe-Operator '|' an die Pipe-Methode weitergeleitet. Pipes können eine beliebige Anzahl an optionalen Parametern akzeptieren, welche mittels einem Doppelpunkt ':' getrennt an die Pipe-Methode

übergeben werden. Zum Beispiel verfügt die DatePipe über einen optionalen Parameter, der das Format des Datums angibt. Zusätzlich können mehrere Pipes miteinander verkettet werden, indem sie nacheinander mittels dem Pipe-Operator angeführt werden. (Angular-Team, 2016)

```
import { Component } from '@angular/core';

@Component({
  template:
    `<p>The item was created on {{ created | date:"fullDate" }}</p>`
})
export class ItemComponent {
  created = new Date(2015, 9, 21);
}
```

Listing 4.12: Beispiel für die Anwendung von integrierten Pipes in Angular [Code adaptiert von Angular-Team (2016)]

Listing 4.13 illustriert die Implementierung einer benutzerdefinierten Pipe namens 'ExponentialPipe'. Die Pipe erhöht den Wert exponentiell, wobei der Exponent als Parameter der Pipe übergeben werden kann. Ist kein Parameter angegeben, wird der Standardwert '1' als Exponent verwendet. Damit Angular weiß, dass es sich dabei um eine Pipe handelt, werden der Klasse zusätzliche Metadaten in Form des Pipe-Decorator hinzugefügt. Die zwingend notwendige 'name'-Eigenschaft gibt dabei den Namen der Pipe an, auf den später deklarativ in den Templates verwiesen werden kann. Die Pipe-Klasse hat die 'PipeTransform'-Schnittstelle zu implementieren, welche eine 'transfrom'-Methode angibt mit einem Input-Wert und zusätzlich optionalen Parametern. Als Rückgabewert wird der transformierte Wert erwartet. Jedes zusätzliche Argument innerhalb der 'transform'-Methode korreliert zu einem Parameter, welcher der Pipe übergeben werden kann. In dem Beispiel der ExponentialPipe existiert genau ein solcher Parameter, der Exponent. Damit die benutzerdefinierte Pipe in der Applikation verwendet werden kann, muss sie zuerst im AppModule zur 'declarations'-Liste hinzugefügt werden. Grundsätzlich können benutzerdefinierte Pipes, sobald sie definiert wurden, auf dieselbe Art und Weise wie integrierte Pipes in Angular angewendet werden. (Angular-Team, 2016)

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'exponential'})
export class ExponentialPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Listing 4.13: Implementierung einer benutzerdefinierten Pipe in Angular [Code adaptiert von Angular-Team (2016)]

Pipes werden in Angular in reine (pure) und unreine (impure) Pipes kategorisiert. Standardmäßig sind alle Pipes pure, solange sie nicht dezidiert als impure innerhalb der Metadaten definiert wurden. Angular führt Pure-Pipes nur aus, wenn eine reine Änderung des Eingangswertes detektiert wurde. Dabei wird unter einer reinen Änderung entweder die Änderung eines primitiven Eingangswertes (String, Number, Boolean, Symbol) oder die Änderung einer Objektreferenz (Date, Array, Function, Object) verstanden. Dagegen ignoriert Angular bei pure-Pipes die Änderungen innerhalb zusammengesetzter Objekte. Zum Beispiel wird die pure-Pipe nicht ausgeführt, wenn einer Liste ein Element hinzugefügt wird oder eine Eigenschaft eines Objektes aktualisiert wurde. Der Grund für diese Restriktion von pure-Pipes liegt in der Performance. Eine Überprüfung lediglich der Objektreferenz ist um ein vielfaches schneller als eine

rekursive, tiefe Überprüfung aller Eigenschaften des Objektes. Angular kann deshalb schnell feststellen, ob eine Ausführung der Pipe und damit auch eine Aktualisierung der View übersprungen werden kann. Aus diesem Grund sollten pure-Pipes immer bevorzugt werden, in manchen Fällen ist dies jedoch nicht möglich und Angular bietet deshalb auch die Möglichkeit Pipes als impure zu deklarieren. Impure-Pipes werden nach jedem Änderungszyklus einer Komponente durch Angular ausgeführt. Das bedeutet, dass solche Pipes nach jedem Tastendruck oder nach jeder Mausbewegung des Benutzers, also sehr häufig, aufgerufen werden. Bei der Implementierung von impure-Pipes in Angular sollte daher, in Anbetracht dieser Tatsache, immer mit großer Vorsicht vorgegangen werden. Eine ausführliche, zeitintensive impure-Pipe kann die Benutzererfahrung erheblich mindern. (Angular-Team, 2016)

4.7 Services

Services ist eine weitläufige Kategorie innerhalb von Angular. Sie umfasst alle Werte, Funktionen oder auch Features, welche die Applikation benötigt. Fast alles kann dabei als Service definiert werden. Typischerweise handelt es sich dabei jedoch um eine Klasse mit einem klar definierten Zweck. Logging-Service, Daten-Service oder die Applikations-Konfiguration sind Beispiele für Services in Angular. Dabei hat Angular selber keine konkrete Definition für einen Service. Es existiert weder eine Basis-Klasse noch eine übergeordnete Stelle, bei der sich ein Service registrieren müsste. Nichtsdestotrotz sind Services ein wichtiges Konzept in Angular und werden typischerweise oft von Komponenten verwendet. Eine bewährte Praxis ist dabei, die Komponenten-Klasse so einfach wie möglich zu halten und komplexere Aufgaben an speziell dafür erstellte Services auszugliedern. Die Aufgabe der Komponente besteht darin als Mediator zwischen der View (Angular-Template) und der Applikationslogik zu dienen. Sie stellt dabei Eigenschaften und Methoden für die Datenbindung zur Verfügung. Alle anderen Aufgaben sollten Komponenten im Idealfall an Services delegieren. Dieses Prinzip wird in Angular nicht erzwungen, aber das Framework unterstützt Entwickler dabei es einzuhalten, indem eine einfache Bereitstellung von Services in Komponenten mittels Dependency-Injection zur Verfügung gestellt wird. (Angular-Team, 2016)

4.8 Dependency-Injection

Dependency-Injection (DI) ist ein wichtiges Entwurfsmuster in der Softwareentwicklung. Angular verfügt über ein eigenes DI-Framework und ermöglicht damit neue Instanzen von Klassen mit all deren Abhängigkeiten zur Verfügung zu stellen. Bei den meisten Abhängigkeiten handelt es sich dabei um Services, welche im vorherigen Kapitel 4.7 behandelt wurden. DI wird in Angular verwendet, um neue Komponenten mit deren benötigten Services zu erstellen. Dabei erkennt Angular an den Parametern des Konstruktors der Komponenten-Klasse, welche Services die Komponente benötigt. Sobald Angular eine Komponente erstellt, werden die benötigten Services bei einem sogenannten Injector angefordert. Der Injector ist eine der Hauptkomponenten innerhalb des DI-Entwurfmusters. Seine Aufgabe besteht darin, einen Container an Service-Instanzen zu verwalten. Ist eine angeforderte Service-Instanz nicht bereits im Container, erzeugt der Injector eine neue Instanz und fügt sie dem Container hinzu, bevor sie an Angular übergeben wird. Sobald alle angeforderten Services mittels dem Injector aufgelöst wurden, ruft Angular den Konstruktor der Komponenten-Klasse mit den Services als Argumente auf. Dieser Prozess ist schematisch in Abbildung 4.5 illustriert und zeigt vereinfacht den DI-Mechanismus in Angular. Damit der Injector weiß, wie eine neue Instanz eines Service erzeugt werden kann, muss zuvor ein sogenannter Provider mittels dem Injector registriert werden. Ein Provider kann als Rezept angesehen werden, welches benötigt wird, um den Service zu erstellen, üblicherweise handelt es sich dabei um die Service-Klasse an sich. Providers können auf Modul- oder auf Komponenten-Ebene registriert werden. Häufig werden Providers innerhalb des Root-Moduls registriert, somit kann auf die gleiche Instanz des Service überall in der Applikation zugegriffen werden. (Angular-Team, 2016)

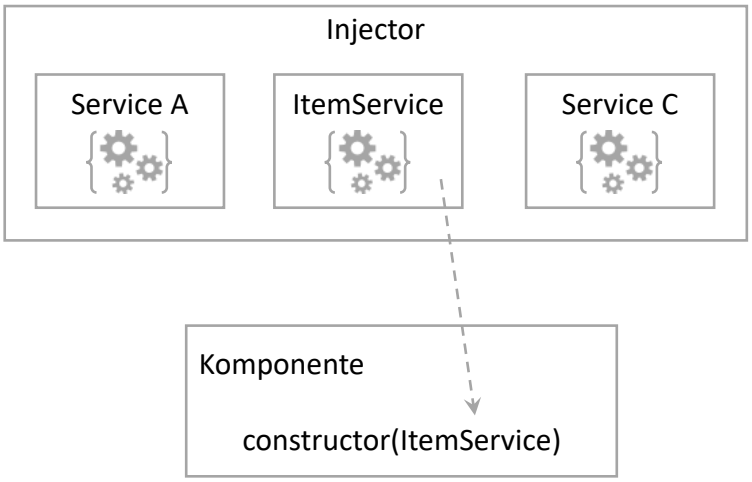

Abbildung 4.5: Vereinfachte Darstellung des Dependency-Injection-Mechanismus in Angular [Abbildung adaptiert von Angular-Team (2016)]

Alternativ können Provider auch auf Komponenten-Ebene registriert werden, indem sie in den Metadaten der Komponente - genauer in der 'providers'-Liste - hinzugefügt werden. Die Registrierung auf Komponenten-Ebene bedeutet, dass für jede neue Instanz der Komponente auch eine neue Instanz des Services erzeugt wird. Auf den entsprechenden Service kann nur die Komponente und deren Kind-Komponenten zugreifen. Technisch wird ein neuer Injector für die jeweilige Komponente erzeugt. Eine Angular-Applikation kann also über mehrere Injector-Instanzen verfügen, welche auf unterschiedlichen Ebenen innerhalb des Komponenten-Baumes der Applikation arbeiten. Fordert eine Komponente einen Service an, versucht Angular zuerst die Nachfrage mittels eines Providers innerhalb der eigenen Injector-Instanz nachzukommen. Fehlt dem Injector auf Komponenten-Ebene der angeforderte Provider, wird die Anfrage an den Injector der Eltern-Komponente weitergereicht. Dieser Prozess wird solange weitergeführt, bis ein Injector mit dem passenden Provider oder das Root-Module erreicht wird. Ist auch im Root-Module der angeforderte Provider nicht registriert, meldet sich Angular mit einer Fehlermeldung. Die Möglichkeit, einen oder mehrere Provider auf unterschiedlichen Ebenen der Applikation zu registrieren, ermöglicht Entwicklern die Abbildung nützlicher Szenarien. Zum Beispiel kann dadurch eine Isolation von Services realisiert werden, weil nur die jeweilige Komponente und deren Kind-Komponenten auf den Service zugreifen können. Das kann die Wartbarkeit der gesamten Anwendung verbessern und damit einhergehend eine Reduzierung von möglichen Fehlern ermöglichen. Das Ersetzen des Service durch eine spezialisierte Implementation des Service tiefer im Komponentenbaum ist ein weiteres Beispiel für die Registrierung von Providern auf unterschiedlichen Komponentenebenen. (Angular-Team, 2016)

4.9 Lifecycle-Hooks

Komponenten und im allgemeinen Direktiven verfügen über einen Lebenszyklus, welcher von Angular verwaltet wird. Angular stellt dabei sogenannte Lifecycle-Hooks bereit, mittels deren Entwickler auf Ereignisse im Lebenszyklus reagieren können. Direktiven haben grundsätzlich die gleichen Lifecycle-Hooks wie Komponenten - mit Ausnahme von Komponenten- oder View-spezifischen Hooks. Entwickler können in Angular auf Lifecylce-Hooks reagieren, indem sie das entsprechende Interface implementieren. Dabei hat jedes Interface eine einzelne Hook-Methode deren Name dem des Interfaces entspricht mit dem Prefix 'ng'. Zum Beispiel verfügt das OnInit-Interface über die ngOnInit-Methode, welche kurz nach dem Erstellen der Komponente aufgerufen wird. Nicht alle Direktiven oder Komponenten imple-

mentieren üblicherweise alle Lifecycle-Hooks. Angular ruft nur Hook-Methoden in Direktiven/Komponenten auf, welche vorher auch definiert wurden. Die Tabelle 4.2 listet alle Lifecycle-Hook-Methoden, sortiert nach der Sequenz in der sie von Angular aufgerufen werden, auf. (Angular-Team, 2016)

Hook	Zweck und Timing
ngOnChanges	Zur Reaktion auf Änderungen von datengebundenen Input-Eigenschaften. Der Methode wird ein SimpleChange-Objekt übergeben - mit dem aktuellen und dem vorherigen Wert der Input-Eigenschaft. Wird vor der ngOnInit-Methode und bei jeder Änderung eines oder mehrerer Input-Eigenschaften aufgerufen.
ngOnInit	Zur Initialisierung der Direktive/Komponente, nachdem alle Input-Eigenschaften gesetzt wurden. Wird einmalig nachdem ersten Aufruf von ngOnChanges aufgerufen.
ngDoCheck	Zur Erkennung und zur Reaktion auf Änderungen, welche Angular nicht selbst erkennt. Wird nach jedem 'change-detection-run', also nach jeder Überprüfung auf Änderungen durch Angular und sofort nach ngOnChanges und ngOnInit aufgerufen.
ngAfterContentInit	Zur Reaktion nachdem Angular externen Inhalt in die Komponenten-Ansicht projiziert hat. Wird einmalig nachdem ersten Aufruf von ngDoCheck aufgerufen. Dieser Hook betrifft nur Komponenten.
ngAfterContentChecked	Zur Reaktion nachdem Angular den externen Inhalt, welcher in die Komponente projiziert wurde, überprüft hat. Wird nach ngAfterContentInit und nach jedem darauffolgenden ngDoCheck aufgerufen. Dieser Hook betrifft nur Komponenten.
ngAfterViewInit	Zur Reaktion nachdem Angular die Komponenten-Ansicht und deren Kind-Komponenten-Ansichten initialisiert hat. Wird einmalig nachdem ersten ngAfterContentChecked aufgerufen. Dieser Hook betrifft nur Komponenten.
ngAfterViewChecked	Zur Reaktion nachdem Angular die Komponenten-Ansicht und deren Kind-Komponenten-Ansichten überprüft hat. Wird nach ngAfterViewInit und nach jedem darauffolgenden ngAfterContentChecked aufgerufen. Dieser Hook betrifft nur Komponenten.
ngOnDestroy	Zum Aufräumen bevor die Direktive/Komponente durch Angular zerstört wird. Typischerweise werden hier zum Beispiel Observables abgemeldet oder Events getrennt, sodass keine Speicherlecks entstehen. Wird kurz bevor die Direktive/Komponente zerstört wird aufgerufen.

Tabelle 4.2: Lifecycle-Hooks, sortiert nach der Sequenz in der sie von Angular aufgerufen werden
[Tabelle adaptiert von Angular-Team (2016)]

4.10 Routing und Navigation

Angular verfügt über einen eigenen Router, mit dessen Hilfe die Navigation innerhalb der Applikation ermöglicht wird. Er kann eine eingegebene URL in den Browser interpretieren und navigiert zur entsprechenden, auf dem Client generierten Ansicht. Dabei können optionale Parameter an die spezifische Ansicht übergeben werden, sodass diese anhand der Parameter entscheiden kann, welcher Inhalt dargestellt werden soll. Der Router kann auch an Links gebunden werden, damit er zur gewünschten Ansicht navigiert, sobald Benutzer mit dem Link reagieren. Die Navigation kann aber auch durch Klicken auf eine Schaltfläche, durch Auswahl in einer Drop-Box oder durch jede andere Reaktion auf Ereignisse erfolgen. Weiter loggt der Router automatisch jede Aktivität im Verlauf des Browsers mit, sodass die Funktionalität der Vor- und Zurück-Schaltfläche des Browsers gewährleistet ist. (Angular-Team, 2016)

Jede Angular-Applikation besitzt genau eine Router-Instanz. Sobald sich die URL in der Adresszeile ändert, sucht der Router nach der entsprechenden Route, in welcher festgelegt ist, welche Komponente angezeigt werden soll. Die Konfiguration der Routes erfolgt mittels der 'RootModule.forRoot'-Methode und Hinzufügen zur 'imports'-Liste des Root-Moduls. Listing 4.14 zeigt beispielhaft eine Konfiguration mit vier Route-Definitionen. Jede dieser Route stellt einen URL-Pfad zu einer Komponente dar. (Angular-Team, 2016)

```
const routes: Routes = [
  { path: 'start', component: StartComponent },
  { path: 'item/:id', component: ItemDetailComponent },
  {
    path: 'items',
    component: ItemListComponent,
    data: { title: 'Items List' }
  },
  { path: '',
    redirectTo: '/items',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(routes)
  ],
  ...
})
export class AppModule { }
```

Listing 4.14: Beispiel-Konfiguration des Router-Service in Angular [Code adaptiert von Angular-Team (2016)]

Dabei können entweder relative oder absolute Pfade angegeben werden. Der Router analysiert den Pfad und stellt zusammen mit dem Base-Pfad, welcher in der index.html mittels des Tags <base> angegeben werden kann, einen endgültigen Pfad zusammen. Der Doppelpunkt in der zweiten Route-Definition gibt einen Parameter mit dem Namen 'id' an. Zum Beispiel wäre bei der URL '/item/42', '42' der Wert des Id-Parameters. Die dazugehörige Komponente 'ItemDetailComponent' kann auf diesen Parameter zugreifen und anhand dessen den passenden Datensatz finden und anzeigen lassen. Die 'data'-Eigenschaft, wie sie in der dritten Route-Definition ersichtlich ist, kann dafür verwendet werden, beliebige Daten, welche mit der Route assoziiert sind, zu speichern. Beispielsweise kann diese Eigenschaft dafür benutzt werden, den Seitentitel, einen Navigationstext oder jeden anderen, schreibgeschützten, statischen Wert

festzulegen. Die leere Pfadangabe in der vierten Route-Definition stellt den Standardpfad der Applikation dar. An diese Route wird der Router navigieren, wenn keine URL angegeben wurde, was üblicherweise beim Start der Applikation der Fall ist. In dem Beispiel 4.14 wird der Standardpfad nach '/items' umgeleitet und deshalb die Komponente 'ItemListComponent' angezeigt. Bei den Zeichen '**' in der letzten Route-Definition handelt es sich um eine sogenannte Wildcard. Der Router wird diese Definition verwenden, wenn keine andere vorherige Definition in der Konfiguration mit der URL übereinstimmt. Eine solche Wildcard-Definition ist praktisch, um '404'-Fehlermeldungen, also eine Fehlermeldung für Seiten, welche nicht gefunden wurden, auszugeben. Der Router verfolgt eine 'Erster-Treffer-Gewinnt'-Strategie, das heißt, spezifische Routen sollten daher vor allgemeinen Routen definiert werden. In der Konfiguration aus dem Beispiel 4.14 werden deshalb zuerst statische Pfade, danach ein leerer Pfad als Standardpfad und zum Schluss ein Wildcard-Pfad definiert. Dieser sollte erst als letzte Möglichkeit vom Router ausgewählt werden. (Angular-Team, 2016)

4.11 RxJS - Reactive Extension for JavaScript

RxJS ist eine Bibliothek, welche es erlaubt, Sequenzen (Streams) von Ereignissen (Events) und Daten zu manipulieren. Dadurch wird die ansonsten komplexe und aufwändige Entwicklung von asynchronen Vorgängen erheblich vereinfacht und verständlicher. Die Bibliothek wird in Angular als Abhängigkeit hinzugefügt und wird für die Verwaltung von Daten-Streams und den Kontrollfluss von Aktionen eingesetzt. RxJS ist eine Kombination aus dem 'Observer'-Entwurfsmuster, dem 'Iterator'-Entwurfsmuster und funktionaler Programmierung. Es ermöglicht eine einfache Erstellung von Event- oder Daten-Streams. Spezielle Abfrage-Operatoren erlauben diese Streams deklarativ zu kombinieren oder zu transformieren. Dabei werden komplexe und aufwändige Aufgaben wie Threads, Thread-Sicherheit, Synchronisation, gleichzeitige Datenstrukturen oder nicht-blockierende I/O-Operationen abstrahiert, damit sich Entwickler auf wesentliche Aspekte innerhalb der Applikation konzentrieren können. Tabelle 4.3 veranschaulicht unterschiedliche Datenzugriff-Modelle. Observables füllen dabei die Lücke für den asynchronen Zugriff auf Sequenzen mehrerer Elemente. Aufgrund der vielen Vorteile wie Kombinierbarkeit, Flexibilität und Vielseitigkeit von Observables werden sie in Angular-Applikationen häufig eingesetzt. (ReactiveX-Team, 2016)

	einzelnes Element	mehrere Elemente
synchron	T getData()	Iterable<T> getData()
asynchron	Future<T> getData()	Observable<T> getData()

Tabelle 4.3: Observables füllen die Lücke für den asynchronen Zugriff auf Sequenzen mehrerer Elemente [Tabelle adaptiert von ReactiveX-Team (2016)]

Frontend-Entwicklung ist immanent asynchron und in der Vergangenheit mangelte es an einem Konzept, welches es erlaubt, die Entwicklung von Benutzeroberflächen in einer funktionalen Art und Weise zu programmieren. Bei dem neuen asynchronen Programmierung-Konzept handelt es sich um den sogenannten Stream, also eine zeitliche Abfolge von Werten. Dabei kann es sich um numerische Werte aber auch zum Beispiel um eine Sequenz von Mausklick-Ereignissen mit den dazugehörigen Koordinaten handeln. Alle Handlungen innerhalb des Browsers können nach dieser Definition als Stream betrachtet werden: Die Abfolge von Browser-Ereignissen, wenn Benutzer mit der Applikation interagieren, das Eintreffen von Daten vom Server oder wenn Zeitlimits ausgelöst werden. (Ferreira, 2016)

Damit Streams vernünftig bei der Entwicklung von Applikationen eingesetzt werden können, ist eine neue asynchrone Primitive - eine Observable - notwendig. Observables ermöglichen die Erstellung, die Kombination, die Manipulierung und das Abonnieren von Streams und sind vergleichbar mit Arrays in einer synchronen Umgebung. Die Kombination aus Stream und einer Menge an funktionalen Operatoren

zur Transformation definiert das Konzept von Observables. RxJS implementiert die Observable-Primitive speziell für JavaScript. Listing 4.15 illustriert, wie ein numerischer Stream, welcher Zahlenwerte von 0 bis 7 im Intervall von zwei Sekunden ausgibt, mittels RxJS erstellt wird. (Ferreira, 2016)

```
let observable = Rx.Observable.interval(2000).take(8);
```

Listing 4.15: Erstellung eines numerischen Streams mittels RxJS [Code adaptiert von Ferreira (2016)]

Observables sind vorgesehen in den ECMAScript-7 - den zukünftigen JavaScript-Standard - aufgenommen zu werden, bis dahin kann RxJS als sogenanntes Pollyfill für Function Reactive Programming (FRP) verwendet werden. Bei FRP handelt es sich um ein Softwareentwicklungs-Paradigma, welches besagt, dass eine komplette Applikation einzig mittels Streams erstellt werden kann. Wird dieses Paradigma angewendet, besteht die Aufgabe hauptsächlich darin, Streams zu erstellen oder Streams an Werten zu identifizieren, an denen die Applikation sich interessiert, sie zu kombinieren und schlussendlich zu abonnieren um auf neue Werte zu reagieren und eine neue Reaktion auszulösen. Die grundsätzliche Idee hinter FRP ist, Programme lediglich deklarativ zu erstellen, indem definiert wird, was Streams sind, wie sie untereinander verlinkt sind und was passieren soll, wenn neue Werte innerhalb eines Stream über die Zeit eintreffen. Das führt dazu, dass solche Programme mit wenigen oder im Idealfall mit keinen Zustands-Variablen auskommt, welche häufig eine Fehlerquelle darstellen. Das bedeutet nicht, dass die Applikation über keinen Zustand verfügt, aber der Zustand ist typischerweise in speziellen Streams oder im DOM und nicht im eigentlichen Source-Code der Applikation gespeichert. (Ferreira, 2016)

Eine wichtige Eigenschaft von Observables ist, dass sie entweder als 'hot' oder 'cold' definiert werden können. Standardmäßig sind sie 'cold' deklariert, das heißt, sie generieren keine neuen Werte, solange sich keine Abonnenten (Subscriber) für das Observable angemeldet haben. Eine weitere wichtige Eigenschaft von Observables ist, dass sie standardmäßig nicht zwischen mehreren Abonnenten geteilt werden. Sobald ein Observable abonniert wird, wird automatisch eine komplett neuer Prozessablauf generiert. Diese beiden Eigenschaften sollten beim Umgang mit Observables stets beachtet werden. (Ferreira, 2016)

In RxJS existieren viele funktionale Operatoren, welche für die Kombination und Manipulation von Observables verwendet werden können. Einige fundamentale Operatoren sind an dieser Stelle nicht abschließend aufgelistet:

- map:
  Transformiert Werte eines Observables in ein anderes Observable.
  Häufige Anwendungsfälle sind: Konvertierung, Parsing, Hinzufügen neuer Felder, etc.

- filter:
  Filtert Werte, welche vom Observable emittiert werden, indem die Abfolge nur mit Werten fortgesetzt wird, welche den Filter-Handler passieren.

- startWith:
  Setzt den initialen Wert für den vorherigen Vorgang, bevor die Abfolge fortgesetzt wird.

- flatMap:
  Nützlich für den Umgang mit einer Menge an Observables.

(Goldshtein, 2016)

4.12 Sicherheitsaspekte in Angular

Angular verfügt über integrierte und bewährte Schutzmechanismen gegen verbreitete Sicherheitslücken und Angriffe auf Webapplikationen wie zum Beispiel Cross-Site Scripting (XSS)-Attacken. Die Implementierung des Schutzes auf Applikationsebene, insbesondere die Authentifizierung und die Autorisierung von Benutzern, ist in den meisten Fällen sehr individuell und liegt daher nicht in der Verantwortung des Frameworks. Damit die Sicherheit von Angular-Applikation gewahrt ist, empfiehlt Angular folgende bewährte Vorgehensweisen:

- Verwendung des aktuellen Angular-Releases:
  Angular aktualisiert in regelmäßigen Abständen seine Bibliotheken. Diese Aktualisierungen reparieren häufig auch Sicherheitslücken, welche in vorherigen Versionen erkannt worden sind.
- Keine Modifizierung des Angular-Codes:
  Private, benutzerdefinierte Versionen von Angular tendieren dazu, nicht mehr regelmäßig aktualisiert zu werden und fallen daher hinter das aktuelle Angular-Release zurück. Das kann dazu führen, das wichtige Änderungen oder Erweiterungen zur Schließung von Sicherheitslücken in der modifizierten Version nicht inkludiert sind.
- Vermeidung von Angular-APIs, welche in der Dokumentation als Sicherheitsrisiko oder als veraltet markiert wurden.

(Angular-Team, 2016)

4.12.1 XSS - Cross-Site-Scripting

XSS-Attacken sind einer der häufigsten Angriffe im Internet und ermöglichen Angreifern, bösartigen Code in Webapplikationen zu injizieren. Mittels solchem Code können zum Beispiel sensible Benutzerdaten gestohlen und Aktionen im vertrauenswürdigen Kontext des betroffenen Benutzers ausgeführt werden. Zur Verhinderung von XSS-Attacken muss sichergestellt sein, dass bösartiger Code nicht in das DOM eingefügt werden kann. Sobald ein Angreifer es schafft, zum Beispiel ein '<script>'-Tag ins DOM einzufügen, kann er beliebigen Code innerhalb der Applikation ausführen. Dabei ist der Angriff nicht nur auf das '<script>'-Tag begrenzt, viele Elemente innerhalb des DOM erlauben, dass Code ausgeführt wird. (Angular-Team, 2016)

Angulars Sicherheitsmodell zur systematischen Verhinderung von XSS-Attacken behandelt zunächst alle Werte standardmäßig als nicht vertrauenswürdig. Jedes Mal bevor ein Wert ins DOM von einem Template über eine Eigenschaft, Attribute, Style oder Datenbindung eingefügt wird, überprüft und entfernt Angular nicht vertrauenswürdige Werte. Angular-Templates sind vergleichbar mit ausführbarem Code: HTML und Datenbindung-Statements (nicht deren Werte zu denen eine Bindung besteht) in Templates werden von Angular als vertrauenswürdig eingestuft. Das bedeutet, dass Benutzereingaben niemals mit Template-Sourcecode verknüpft werden dürfen. Angular stellt für diese Art von Angriffen, welche 'Template-Injection' genannt werden, einen Offline-Template-Compiler zur Verfügung, welcher Template-Injections effektiv verhindert. Als positiver Nebeneffekt steigert der Offline-Template-Compiler gegenüber dynamisch generierter Templates die Performance der Applikation beträchtlich. (Angular-Team, 2016)

Listing 4.16 illustriert beispielhaft, wie die Sanitization in Angular funktioniert und automatisch potenziell schädlichen Code entfernt wird. Im dargestellten Template wird eine Datenbindung zum Wert der Eigenschaft 'htmlSnippet' erstellt. Zuerst durch interpolieren des Wertes in den Inhalt eines Elementes und danach durch Bindung zu der 'innerHTML'-Eigenschaft. Listing 4.17 zeigt die Ausgabe des

Beispiel-Codes. Dabei ist erkennbar, dass interpolierter Code immer entschlüsselt und nicht interpretiert wird. Der Wert von 'htmlSnippet' wird in diesem Fall als Text dargestellt mit allen HTML-Tags. Soll der Wert interpretiert werden, so muss er wie im zweiten Fall zu einer HTML-Eigenschaft gebunden werden. Dies stellt eine potenzielle XSS-Sicherheitslücke dar, welche Angular erkennt und automatisch zuerst eine sogenannte Sanitization - also das Herausfiltern von potenziell schadhaften Code - durchführt. Angular entfernt wie in Listing 4.17 erkennbar die '<script>'-Tags, behält jedoch sicheren Code, wie den Inhalt der '<script>'-Tags oder das ungefährliche '<b>'-Element. (Angular-Team, 2016)

```
@Component({
  template:
    `<p>Gebundener Wert:</p>
     <p>{{htmlSnippet}}</p>
     <br />
     <p>Ergebnis der Datenbindung zu innerHTML:</p>
     <p [innerHTML]="htmlSnippet"></p>`
})
export class InnerHtmlBindingComponent {
  htmlSnippet = 'Template <script>alert("Hack3d")</script> <b>Fetter Text
      </b>';
}
```

Listing 4.16: Angular-Sanitization Beispiel [Code adaptiert von Angular-Team (2016)]

```
Gebundener Wert:
Template <script>alert("Hack3d")</script> <b>Fetter Text</b>

Ergebnis der Datenbindung zu innerHTML:
Template alert("Hack3d") Fetter Text
```

Listing 4.17: Ausgabe des Angular-Sanitization Beispiels von Listing 4.16 [Code adaptiert von Angular-Team (2016)]

Für den Zugriff zum DOM kann Angular umgangen und direkt die Browser-DOM-API oder andere Drittanbieter-APIs verwendet werden. Diese APIs schützen jedoch nicht automatisch vor Sicherheitslücken und sollten daher vermieden und stattdessen Angular-Templates verwendet werden. (Angular-Team, 2016)

Eine weitere wichtige Vorgehensweise zur Verhinderung von XSS-Attacken ist die Aktivierung der Content Security Policy (CSP). Dabei handelt es sich nicht um ein Angular internes Sicherheitskonzept, sondern um eine vertiefende Methode, welche auf dem Webserver konfiguriert werden muss, um einen entsprechenden CSP-Header zurückzugeben. (Angular-Team, 2016)

4.12.2 XSRF - Cross-Site-Request-Forgery

Bei Cross-Site Request Forgery (XSRF) handelt es sich um eine Sicherheitslücke auf HTTP-Ebene. Solche Sicherheitslücken müssen primär auf dem Server entschärft werden, jedoch verfügt Angular über einen integrierten Schutzmechanismus und Tools, welche die Integration auf dem Client vereinfachen. Bei einer XSRF versuchen Angreifer, die Benutzer auf eine andere Webseite (zum Beispiel eval.com) mit bösartigem Code zu leiten. Der Code setzt dann im Hintergrund versteckt eine Anfrage an den Webserver der Applikation (zum Beispiel my-app.com) ab. Denkbar wäre zum Beispiel ein Szenario, indem ein Benutzer bereits in der Applikation (my-app.com) angemeldet ist. Gleichzeitig öffnet der Benutzer ein Email und klickt auf einen Link zu 'eval.com', welcher in einem neuen Tab/Fenster im Browser geöffnet wird. Die 'eval.com'-Webseite sendet sofort eine bösartige Anfrage an 'my-app.com'. Je nach

Art der Anwendung könnte es sich zum Beispiel um eine Transaktion handeln. Der Browser sendet automatisch die 'my-app.com'-Cookies inklusive dem Authentifizierungs-Cookie mit der Anfrage an den Server. Verfügt der entsprechende Server über keine XSRF-Sicherungsmaßnahmen, kann er nicht zwischen berechtigten und gefälschten Anfragen unterscheiden. (Angular-Team, 2016)

Damit solche Attacken verhindert werden können, muss die Applikation dafür sorgen, dass Benutzeranfragen nur von der wirklichen Applikationsseite und nicht von anderen Webseiten stammen. Der Server und der Client müssen dazu zusammenarbeiten und sie können nur gemeinsam XSRF-Attacken abwehren. Eine verbreitete Abwehrmaßnahme gegen XSRF besteht darin, dass der Server einen zufällig generierten Authentifizierung-Token in einem Cookie an den Client sendet. Der Client liest das Cookie und fügt einen benutzerdefinierten Header mit dem Token in allen nachfolgenden Anfragen hinzu. Der Server vergleicht den Wert des Cookies mit dem Wert im Header der Anfrage und verweigert die Anfrage sobald die Werte fehlen oder nicht übereinstimmen. Diese Technik ist effektiv, weil alle Browser die sogenannte 'same origin policy' implementieren. Das bedeutet, dass nur Code von der gleichen Webseite, welche die Cookies auch gesetzt haben, diese wiederum lesen können und benutzerdefinierte Header für Anfragen ebenfalls nur von dieser Seite gesetzt werden können. Wird die 'same origin policy' richtig umgesetzt, kann nur die Applikation und nicht der bösartige Code von 'evil.com' das Cookie-Token lesen und den benutzerdefinierten Header setzen. (Angular-Team, 2016)

Der 'http'-Service in Angular unterstützt den clientseitigen Teil dieser im vorherigen Absatz erläuterten Abwehrmaßnahme gegen XSRF-Attacken. Standardmäßig ist dazu die sogenannte 'CookieXSRFStrategy' aktiviert. Bevor eine HTTP-Anfrage gesendet wird, sucht die 'CookieXSRFStrategy' nach einem Cookie mit dem Namen 'XSRF-TOKEN' und setzt einen Header namens 'X-XSRF-TOKEN' mit dem Wert aus dem Cookie. Der Server muss dementsprechend zuerst das 'XSRF-TOKEN'-Cookie setzen und bei jeder nachfolgenden Anfrage bestätigen, dass ein 'XSRF-TOKEN'-Cookie und ein 'X-XSRF-TOKEN'-Header vorhanden und übereinstimmen. 'XSRF'-Tokens sollten eindeutig pro Benutzer/Session, aus einer großen, zufälligen Zahl bestehen, welche durch einen kryptographisch sicheren Zufallsgenerator erzeugt wurde und nach ein oder zwei Tagen ablaufen. Der Server kann auch andere Namen, als die von Angular standardmäßig definierten Namen für das Cookie und den benutzerdefinierten Header verwenden. Dazu kann die 'CookieXSRFStrategy' in Angular angepasst werden. (Angular-Team, 2016)

4.12.3 XSSI - Cross-Site-Script-Inclusion

Cross-Site Script Inclusion (XSSI), ist auch bekannt als JSON-Schwachstelle und erlaubt es Angreifern, Daten über eine JSON-API auszulesen. Die Attacke funktioniert bei älteren Browsern mittels Überschreiben der nativen JavaScript-Objekt-Konstruktors und Hinzufügen einer API-URL unter Ausnützung des '<script>'-Tags. Eine XSSI-Attacke ist nur erfolgreich, wenn die zurückgegebenen JSON-Daten als JavaScript ausführbar sind. Server können solche Angriffe verhindern, indem sie alle JSON-Rückmeldungen mit der bekannten Zeichenfolge ')]}',\n' voranstellen und sie damit nicht mehr ausführbar machen. Der 'http'-Service von Angular erkennt diese Konvention und entfernt automatisch die Zeichenfolge ')]}',\n' von allen JSON-Rückmeldungen bevor sie weiterverarbeitet werden. (Angular-Team, 2016)

4.13 Testen von Angular-Applikationen

Das Erstellen von Tests ist in der professionellen Softwareentwicklung unentbehrlich und hat deshalb auch in Angular einen hohen Stellenwert. Tests helfen das Verhalten einer Applikation zu erkunden und zu bestätigen und erfüllen im wesentlichen drei Grundfunktionen:

1. Tests schützen vor Veränderungen, welche bestehenden Code unbrauchbar machen (Regression).

2. Tests verdeutlichen, was der Code macht - sowohl bei beabsichtigtem Verhalten als auch bei abweichenden Bedingungen.

3. Test zeigen Fehler in der Architektur und in der Implementation auf, indem sie den Code aus mehreren Winkeln betrachten. Wenn ein Teil einer Applikation schwierig zu testen scheint, liegt die Ursache meist an Designfehlern, welche in der Anfangsphase oft einfacher und mit geringerem Aufwand zu beheben sind, als zu einem späteren Zeitpunkt.

(Angular-Team, 2016)

Tests können in Angular mit einer Vielzahl an Technologien und Tools erstellt werden. Tabelle 4.4 listet eine Auswahl an bewährten Technologien auf, welche von Angular empfohlen werden.

Technologie	Zweck
Jasmine[1]	Das Jasmine-Test-Framework liefert alles was benötigt wird, um grundlegende Tests zu schreiben. Jasmine enthält einen HTML-Test-Runner, mit dessen Hilfe Tests innerhalb eines Browsers ausgeführt werden können.
Angular-Testing-Utilities[2]	Das Angular-Testing-Utilities erstellt eine Testumgebung für den zu testenden Angular-Applikationscode. Die Testumgebung kann verwendet werden, um Teile der Applikation aufzubereiten und zu steuern, während sie mit dem Angular-Framework interagiert.
Karma[3]	Der Karma-Test-Runner ist ideal für das Schreiben und Ausführen von Unit-Tests während der Entwicklung der Applikation. Karma kann ein integraler Bestandteil des Entwicklungs- und des kontinuierlichen Integrationsprozesses des Projekts sein.
Protractor[4]	Protractor wird für das Schreiben und Ausführen von End-to-End-(e2e)-Tests eingesetzt. End-to-End-Tests können die Anwendung im gleichen Maße erkunden, wie Endbenutzer die Anwendung erleben. Während e2e-Tests ausgeführt werden, laufen zwei Prozesse: Ein Prozess führt die reale Anwendung aus und ein zweiter Prozess führt die Protractor-Tests aus, welche die Benutzerinteraktion simulieren und stellen somit sicher, dass die Applikation wie erwartet im Browser reagiert.

Tabelle 4.4: Auswahl an Technologien für das Erstellen von Tests in Angular [Tabelle adaptiert von Angular-Team (2016)]

Isolierte Unit-Test untersuchen eine Instanz einer Klasse ohne deren Abhängigkeiten zu Angular oder andere injizierte Werte (zum Beispiel mittels Dependency-Injection) zu berücksichtigen. Die Testumgebung erstellt eine Testinstanz von der entsprechenden Klasse mit neuen Kopien der Konstruktor-Parametern und führt anschließend die Tests mittels der Test-API aus. Isolierte Unit-Tests sollten speziell für Pipes und Services erstellt werden. Komponenten können ebenfalls in Isolation getestet werden, allerdings können isolierte Unit-Tests nicht aufzeigen, wie Komponenten mit Angular interagieren. Insbesondere können sie nicht zeigen, wie eine Komponenten-Klasse mit ihrem eigenen Template oder anderen Komponenten interagieren. Für diesen Zweck stellt Angular die sogenannte Angular-Testing-Utilities zur Verfügung mit einer 'TestBed'-Klasse und einigen weiteren Helper-Tools. (Angular-Team, 2016)

[1] Jasmine - https://jasmine.github.io/
[2] Angular-Testing-Utilities - https://angular.io/docs/ts/latest/guide/testing.html#!#atu-apis
[3] Karma - https://karma-runner.github.io/
[4] Protractor - http://www.protractortest.org/

4.14 Integration von Angular in Meteor

Meteor ist ein Full-Stack JavaScript-Framework, das sich um viele Aufgaben bei der Erstellung von Webapplikationen oder mobilen Applikationen kümmert: Von der Datenbank, der Kommunikation zwischen Server und Client bis hin zur Unterstützung von mobilen Endgeräten. Es unterstützt Full-Stack Reactivity ohne zusätzliche Komponenten, sodass Änderungen in der Datenbank automatisch bis zum Client in Echtzeit weitergeleitet werden. Darüber hinaus ist Meteor und auch die intern eingesetzten Technologien komplett Open-Source. In Kapitel 3 wurde Meteor im Detail erläutert, an dieser Stelle soll das Zusammenspiel und die Integration zwischen Meteor und Angular genauer betrachtet werden.

Angular kann mittels der Angular-Meteor Bibliothek [5] einfach in eine Meteor-Applikation eingebettet werden. Bei der Erstellung der Bibliothek wurde darauf geachtet, dass auch andere Client-Frameworks, wie zum Beispiel Blaze oder React, in der selben Applikation und sogar innerhalb der gleichen Komponente beziehungsweise Direktive verwendet werden können. Damit sind Entwickler unabhängiger gegenüber einer Technologie und können etwa mit Angular beginnen und später zu einem anderen Framework migrieren. Ein weiterer Vorteil besteht darin, dass Bibliotheken und Lösungen von unterschiedlichen Communities eingesetzt werden können. Es gibt viele Gründe Angular als Client-Framework innerhalb von Meteor einzusetzen, die zentralen Vorteile sind insbesondere:

- Angular ermöglicht eine bewährte MVC- beziehungsweise MVVM-Architektur auf dem Client.
- Es können viele bereits existierende Angular-Applikationen weiterverwendet oder mit geringem Aufwand migriert werden.
- Angular verfügt über ein umfassendes Ökosystem und eine große und aktive Community.
- Für viele Angular-Entwickler ermöglicht die Kombination von Angular und Meteor einen einfachen Einstieg in die Meteor-Plattform
- Alle weiterreichenden Vorteile von Angular 2 gegenüber Angular 1 sind auch in Zusammenspiel mit Meteor gültig.

(Goldshtein, 2016)

Umgekehrt bietet Meteor als umfassendes Full-Stack Framework viele Pluspunkte, wenn es in einer Angular-Applikation eingesetzt wird. Die vordergründigen Vorteile sind vorwiegend:

- Meteor bietet eine Echtzeit-Backend Unterstützung für Angular.
- Sehr einfache, intuitive Integration von reaktiven Datenbanken.
- Meteor umfasst den vollständigen Technologie-Stack innerhalb einer JavaScript-Webapplikation.
- Das Framework und deren eingesetzte Technologie sind genau wie Angular komplett Open-Source.
- Einfaches Deployment von Webapplikationen

(Goldshtein, 2016)

Speziell die Handhabung mit Daten wurde in Angular 2 im Vergleich zu Angular 1 wesentlich weiterentwickelt. Die strukturellen Änderungen ermöglichten es, einen einfachen Datensynchronisations-Layer

[5] Angular-Meteor Bibliothek - https://angular-meteor.com/

für Meteor zu konstruieren. Dadurch ist es einerseits möglich, die nativen APIs von Meteor zu verwenden, wie in anderen herkömmlichen Meteor-Applikationen, und andererseits Angular 2 Code zu schreiben, wie in jeder anderen Angular-Applikation. Das bedeutet, dass Meteor-/Angular-Entwickler auf Lösungen und Wissen der gesamten Meteor-Community zurückgreifen können, egal welches Client-Framework sie verwenden. (Brad Green, 2016)

Meteor setzt standardmäßig Blaze[6] - Meteors eigens entwickeltes Client-Framework - als Template-Engine und Babel[7] als Pre-Prozessor ein. Für die Integration von Angular und Typescript als JavaScript-Compiler muss daher zunächst das Blaze-Packet entfernt und das Packet 'angular2-compilers' hinzugefügt werden. Der Vorgang ist in Listing 4.18 ersichtlich. (Goldshtein, 2016)

```
$ meteor remove blaze-html-templates

$ meteor add angular2-compilers
```

Listing 4.18: Integration von Angular in ein Meteor-Projekt [Code adaptiert von Goldshtein (2016)]

Das 'angular2-compilers'-Packet ersetzt nicht nur Blaze, sondern deklariert automatisch auch Typescript als Script-Sprache im entsprechenden Projekt. Zusätzlich werden alle CSS-Dateien mit dem Pre-Prozessor SASS[8] kompiliert. SASS vereinfacht den Umgang und den Gestaltungsprozess von HTML-Seiten erheblich. Der Typescript-Compiler arbeitet auf Basis einer benutzerdefinierten Konfiguration, welche innerhalb einer JSON-Datei mit dem Namen 'tsconfig.json' deklariert wird. Listing 4.19 veranschaulicht eine typische Typescript-Konfiguration innerhalb einer Meteor Applikation. (Goldshtein, 2016)

```
{
  "compilerOptions": {
    "allowSyntheticDefaultImports": true,
    "baseUrl": ".",
    "declaration": false,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [
      "dom",
      "es2015"
    ],
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "target": "es5",
    "skipLibCheck": true,
    "stripInternal": true,
    "noImplicitAny": false,
    "types": [
      "meteor-typings",
      "@types/underscore"
    ]
  },
  "include": [
    "client/**/*.ts",
    "server/**/*.ts",
    "imports/**/*.ts"
```

[6] Blaze - Meteor Client-Framework - http://blazejs.org/guide/introduction.html
[7] Babel - JavaScript-Compiler - https://babeljs.io/
[8] SASS - CSS Compiler - http://sass-lang.com/

```
  ],
  "exclude": [
    "node_modules"
  ],
  "compileOnSave": false,
  "atom": {
    "rewriteTsconfig": false
  }
}
```

Listing 4.19: Typische Konfiguration von Typescript innerhalb einer Meteor Applikation [Daten extrahiert von Goldshtein (2016)]

Angular ist abhängig von sogenannten 'peer dependencies', welche ebenfalls dem entsprechenden Projekt hinzugefügt werden müssen. Listing 4.20 listet alle typischen Abhängigkeiten auf, welche mittels npm[9] - einem Paketmanager für JavaScript, zur Verwaltung von Abhängigkeiten - installiert werden können. Nach diesen wenigen notwendigen Änderungen in der Standard-Konfiguration einer Meteor-Applikation ist Angular bereits vollständig integriert und es kann mit der Entwicklung der Applikation begonnen werden, üblicherweise durch Erstellung des Root-Moduls und der Root-Komponente, wie es auch in anderen Angular-Applikation ohne Meteor der Fall ist. (Goldshtein, 2016)

```
$ meteor npm install --save @angular/common
$ meteor npm install --save @angular/compiler
$ meteor npm install --save @angular/core
$ meteor npm install --save @angular/forms
$ meteor npm install --save @angular/http
$ meteor npm install --save @angular/platform-browser
$ meteor npm install --save @angular/platform-browser-dynamic
$ meteor npm install --save @angular/platform-server
$ meteor npm install --save meteor-rxjs
$ meteor npm install --save reflect-metadata
$ meteor npm install --save rxjs
$ meteor npm install --save zone.js
$ meteor npm install --save-dev @types/meteor
$ meteor npm install --save-dev @types/underscore
$ meteor npm install --save-dev meteor-typings
```

Listing 4.20: Typische Abhängigkeiten von Angular, welche mittels npm installiert werden können [Code adaptiert von Goldshtein (2016)]

[9] npm - JavaScript-Paketmanager - https://www.npmjs.com/

Kapitel 5

Prototypische Implementierung eines webbasierten Editors zur Erstellung von reaktiven Webapplikationen basierend auf Meteor und Angular

“ When designers replaced the command line interface with the graphical user interface, billions of people who are not programmers could make use of computer technology. ”

[Rheingold, 2012]

Nachdem sich die vorherigen beiden Kapitel mit den Grundlagen von Meteor und Angular befasst haben, widmet sich dieses Kapitel der prototypischen Umsetzung eines webbasierten Editors zur Erstellung von reaktiven Webapplikationen, basierend auf den zuvor erwähnten Frameworks. Dieser webbasierte Editor wurde im Zuge dieser Arbeit eigenständig implementiert, als Nachweis der Machtbarkeit der vorgestellten Konzeptidee, und spiegelt den praktischen Teil dieser vorliegenden Arbeit wider. Der vergebene Arbeitstitel ’webRAD - Rapid Application Development for webApps’ für den Editor soll dabei die zugrunde liegende Intention und Konzeptidee, welche in Kapitel 2.7 dargelegt wurde, ausdrücken. Wie Howard Rheingold bereits im einleitenden Zitat beschreibt, ist eine grafische Benutzeroberfläche notwendig, damit End-User effektiv in den Entwicklungsprozess miteinbezogen werden können, Meteor stellt diesbezüglich jedoch lediglich eine Kommandozeile als Interaktionsschnittstelle zur Verfügung. Der Editor webRAD knüpft bei dieser Problematik an und repräsentiert eine grafische und benutzerfreundliche Abstraktionsebene über den Frameworks Meteor und Angular, welche speziell an die Bedürfnisse von End-Usern ausgerichtet ist. Gleichzeitig ermöglicht webRAD einen kontinuierlichen Entwicklungsprozess, indem darauf geachtet wurde, dass webRAD qualitativen Code generiert, welcher in späteren Projektphasen von professionellen Softwareentwicklern heruntergeladen und wiederverwendet werden kann.

5.1 Aufbau und Funktionsweise

Bei webRAD handelt es sich um eine webbasierte Entwicklungsumgebung, welche die Erstellung von Webapplikationen, basierend auf Meteor und Angular, ermöglicht. Dabei baut die prototypische Implementierung des Editors auf den gleichen Frameworks beziehungsweise Technologien auf, wie die Webapplikationen, welche mittels webRAD generiert werden können. Deshalb impliziert webRAD zum

Beispiel die gleichen reaktiven Eigenschaften, wie andere Meteor-Applikationen. Dieser Aspekt ermöglicht, dass mehrere Benutzer gleichzeitig an demselben Projekt zusammenarbeiten können und jede Änderung eines Benutzers automatisch und in Echtzeit für die jeweils anderen Benutzer sichtbar wird.

Das Design von webRAD ist geprägt durch den Einsatz von Angular Material[1]. Angular Material bietet eine umfangreiche Sammlung an modernen UI-Komponenten, welche sowohl in Webapplikationen als auch in mobilen Applikationen funktionieren und die von Google vorgegebenen Material-Designrichtlinien[2] berücksichtigen. Die Material UI-Komponenten werden vom Angular-Team entwickelt und sind speziell für eine nahtlose Integration in Angular-Anwendungen optimiert.

Der Editor webRAD ist im Wesentlichen in drei Bereiche aufgegliedert: Der Projektverwaltung und innerhalb der Projektansicht in den Daten-Editor und den Screen-Editor, die in den folgenden Unterkapiteln genauer erläutert werden.

5.1.1 Projektverwaltung

Nach der Anmeldung bei webRAD befinden sich die Benutzer in der Projektverwaltung, in der alle Projekte aufgelistet werden, welche von ihnen angelegt oder mit ihnen geteilt wurden. In der Projektverwaltung können Projekte aktualisiert, gelöscht oder neue Projekte, wie in Abbildung 5.1 ersichtlich, hinzugefügt werden. Beim Erstellen eines neuen Projektes muss ein Name und optional eine kurze Beschreibung vergeben werden. Zusätzlich kann ausgewählt werden, ob das neue Projekt öffentlich oder nur für eingeladene Benutzer zugänglich sein soll. Auf ein öffentliches Projekt kann jeder, der über die öffentliche Projekt-URL verfügt, zugreifen.

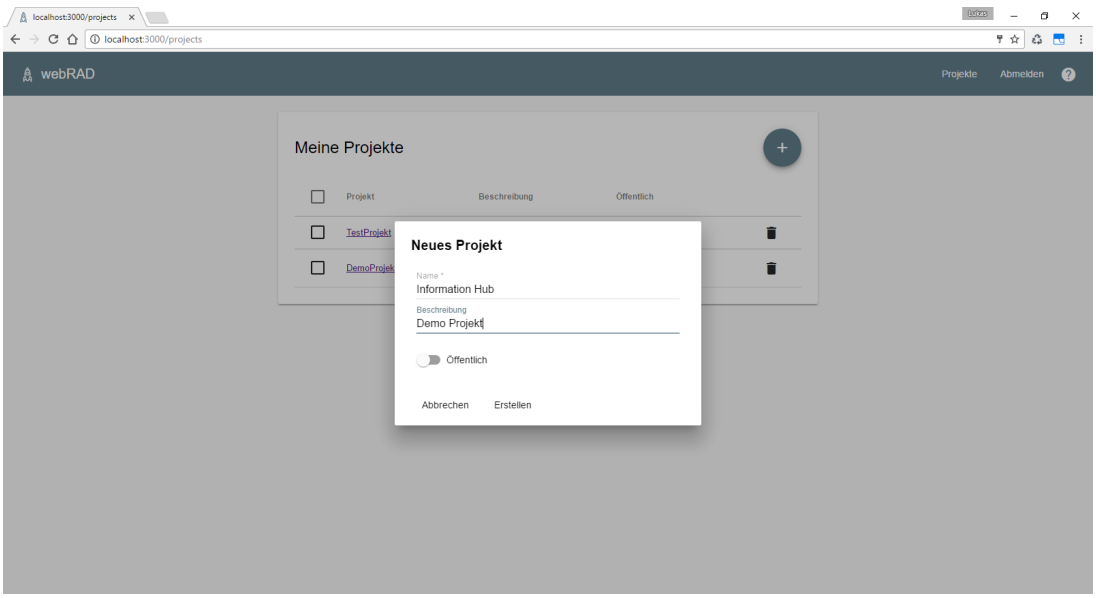

Abbildung 5.1: Screenshot der Projektverwaltung in webRAD, einschließlich dem modularem Dialog, indem ein neues Projekt erstellt werden kann

Während des Erstellungsprozesses eines Projektes erzeugt webRAD automatisch eine Basis-Dateistruktur mit allen erforderlichen Dateien, welche für eine lauffähige Anwendung benötigt werden. Eine typische Meteor-Applikation enthält zumindest ein Server- und ein Client-Verzeichnis mit Skript-Dateien, welche je nachdem in welchem Verzeichnis sie abgelegt sind, nur auf dem Server beziehungsweise nur auf dem

[1] Angular Material - Material-Design-Komponenten für Angular-Anwendungen - https://material.angular.io/
[2] https://material.io/guidelines/

Client geladen werden. Skript-Dateien außerhalb dieser beiden Verzeichnisse werden in beiden Umgebungen geladen und ausgeführt. Die genaue Meteor-Verzeichnisstruktur wird in Kapitel 5.2.2 behandelt.

Standardmäßig wird Blaze als clientseitiges Framework und Template-Engine in Meteor verwendet. Bei der Projekt-Erstellung ersetzt webRAD das Blaze-Framework durch Angular und fügt Typescript als Compiler hinzu. Bei Typescript handelt es sich um eine typisierte Obermenge von JavaScript, welche zu gewöhnlichem JavaScript transpiliert werden kann. Typescript ist die von den Angular-Entwicklern empfohlene Programmiersprache und wird im Kapitel 5.2.1 detailliert erläutert.

Zusätzlich werden während der Erstellung eines Projekts von webRAD alle erforderlichen npm-Pakete, wie zum Beispiel Angular, Material, jQuery oder Typescript heruntergeladen und installiert. Dadurch wird sichergestellt, dass Benutzer anschließend sofort ihr noch überwiegend leeres, aber lauffähiges Projekt starten können.

5.1.2 Projektansicht

Innerhalb eines Projektes ist webRAD, wie in Abbildung 5.2 illustriert, in unterschiedliche Bereiche aufgeteilt. Ganz oben befindet sich das Hauptmenü zur Verwaltung der Live-Vorschau, dem Projekt-Download, dem Deployment, den Projekteinstellungen und für die Navigation zurück zur Projektverwaltung. Auf der linken Seite des Bildschirms ist jeweils eine Listenansicht für Daten-Schemas und Screens (Ansichten) platziert, welche im Projekt deklariert werden. Über die jeweiligen Einträge in den Listenansichten lässt sich der entsprechende Daten-Editor beziehungsweise Screen-Editor im Hauptbereich in der Mitte des Bildschirms öffnen. Dieser Hauptbereich verfügt über ein Tab-Layout, sodass mehrere Editoren gleichzeitig geöffnet werden können. Im unteren Bereich der Projektansicht befindet sich das Konsolenfenster, in dem stets die serverseitige Ausgabe der Anwendung in Echtzeit angezeigt wird. Das Konsolenfenster kann für eine bessere Übersicht verkleinert oder vollständig minimiert werden.

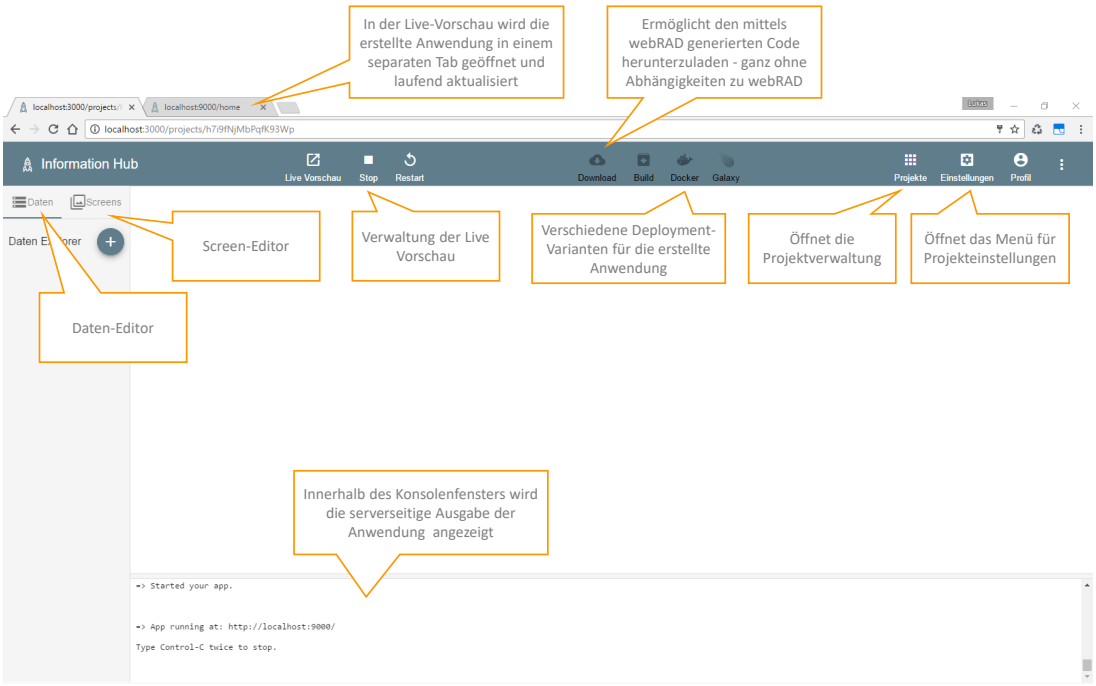

Abbildung 5.2: Screenshot und Illustration des Aufbaus der Projektansicht in webRAD

Die Live-Vorschau ermöglicht die Begutachtung der Fortschritte bei der Entwicklung der Webapplika-

tion. Dazu wird für jedes Projekt eine neue Node-Instanz als Kind-Prozess mit einem freien Port auf dem Server gestartet. Dabei wird berücksichtigt, dass mehrere Benutzer an demselben Projekt arbeiten können und demzufolge in diesen Fällen keine weiteren Instanzen notwendig sind. Die dafür zur Verfügung stehenden Ports auf dem Server werden mittels eines Port-Pools verwaltet, welcher je nach Server-Infrastruktur konfiguriert werden kann. Änderungen innerhalb des Projekts detektiert das Meteor-Framework automatisch und kompiliert daraufhin die entsprechenden Dateien neu. Anschließend werden die betroffenen Clients informiert, damit sie die Anwendung neu laden können. Dieser Vorgang wird auch als 'Hot Code Push' bezeichnet und ermöglicht, dass Benutzer, mit einer geringen Verzögerung, immer das aktuelle Ergebnis beobachten können, ohne selbst aktiv die Webanwendung zu aktualisieren.

5.1.3 Daten-Editor

Der Daten-Editor in webRAD ermöglicht die einfache Erstellung von Datenschemas. Für diesen Zweck generiert webRAD intern automatisch MongoDB-Collections, in denen Daten persistent gespeichert werden können. MongoDB ist im Allgemeinen eine schemafreie, dokumentenorientierte Datenbank. Nichtsdestotrotz ist es möglich, in webRAD ein Schema vorzugeben, welches zur Unterstützung für End-User eine bewährte Vorgehensweise ist. Ein solches Schema ist im Vergleich zu herkömmlichen, relationalen Datenbanken flexibler und skalierbarer. Abbildung 5.3 zeigt beispielhaft, wie ein Datenschema in webRAD aufgebaut werden kann. Ein Datenschema besteht aus einzelnen Feldern mit einem Namen, Anzeigenamen, Typ und einem Indikator, welcher angibt, ob das Feld zwingend erforderlich ist. Für jede Collection generiert webRAD zusätzlich ein Interface, welches anhand der angegebenen Datentypen der Felder aufgebaut wird. Dabei abstrahiert webRAD das Konzept der Datentypen und stellt sie in einer speziell für End-User optimierten Variante zur Verfügung. Die auswählbaren Datentypen wie zum Beispiel Text, Nummerisch, Boolean, Passwort oder URL werden dafür später in die gewöhnlichen Datentypen, welche TypeScript bereitstellt, konvertiert.

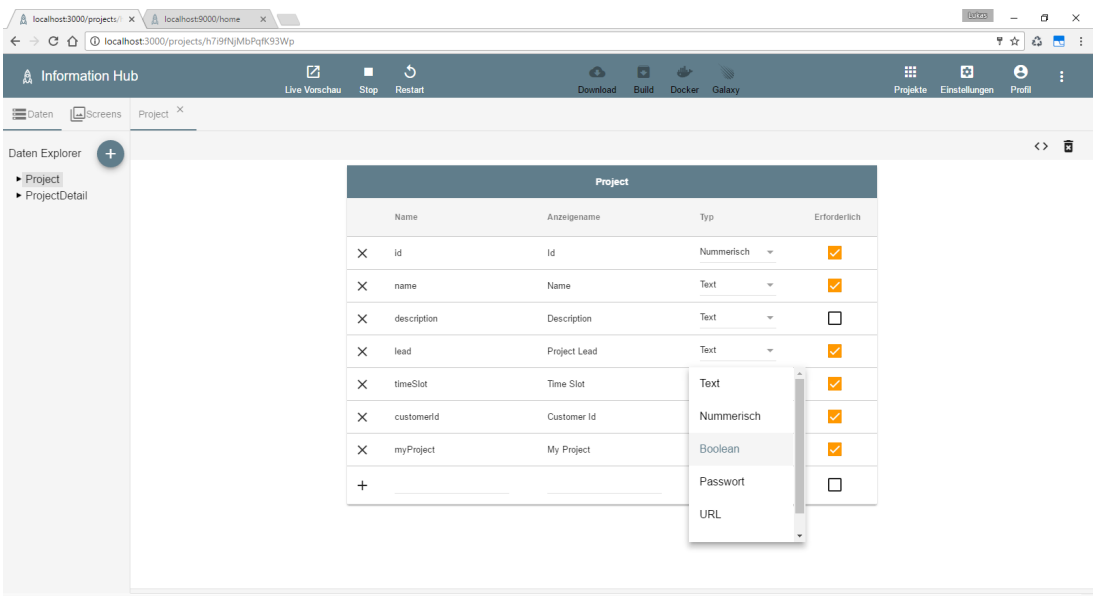

Abbildung 5.3: Screenshot des Daten-Editors in webRAD, der zur Erstellung von Datenschemas dient

Die Erstellung von Interfaces zusätzlich zu Collections ist ein wesentlicher Aspekt für die Generierung von qualitativem Code. Eines der Kernprinzipien von TypeScript besteht darin, dass sich die Typen-Überprüfung auf die Form der entsprechenden Objekte fokussiert. Dieser Vorgang wird häufig auch als 'duck typing' oder 'structural subtyping' bezeichnet. In TypeScript übernehmen Interfaces die Rolle der Benennung von Typen und sie sind eine wirksame Möglichkeit für Kontrakte innerhalb des eigenen

Projekt-Codes, sowie Kontrakte mit Code außerhalb einer Anwendung. (TypeScript-Team, 2017)

Für eine bessere Veranschaulichung und ein leichteres Testen der Anwendung werden meist Dummy-Daten innerhalb der ansonsten zu Beginn leeren Collections benötigt. Aus diesem Grund stellt webRAD für jede Collection eine Tabelle, wie in Abbildung 5.4 erkennbar, zur Verfügung. Diese Tabelle passt sich automatisch an das deklarierte Datenschema an und stellt für die jeweiligen Datentypen die korrespondierenden Eingabe-Elemente bereit.

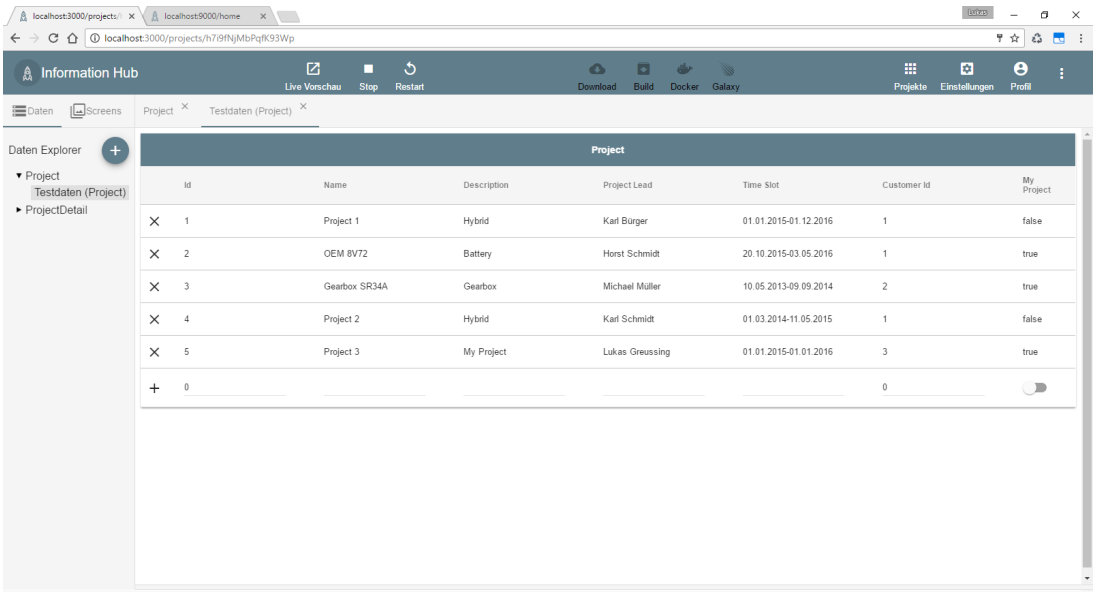

Abbildung 5.4: Screenshot für die Eingabe von Dummy-Daten in webRAD

5.1.4 Screen-Editor

Jeder Screen in webRAD entspricht einer Bildschirm-Ansicht innerhalb der als Single-Page-Applikation (SPA)-konzipierten Webanwendung zwischen denen navigiert werden kann. Intern generiert webRAD für jeden Screen eine eigene Angular-Komponente. Für die Konstruktion der Screens werden in webRAD Elemente, wie in Abbildung 5.5 illustriert, in einer Baumstruktur angeordnet.

Die Elemente innerhalb des Screen-Editors werden in drei Gruppen unterteilt: Layout-, Daten- und UI-Elemente. Diese Elemente können frei mittels Drag & Drop innerhalb der Baumstruktur verschoben werden, beschränkt lediglich dadurch, dass einzig Layoutelemente weitere Kind-Elemente beinhalten können. Sobald ein Element selektiert wurde, werden auf der rechten Seite des Editors die entsprechenden Eigenschaften und Ereignisse angezeigt. Intern entspricht jedes Element einer Angular-Komponente, daher können auf Eigenschaften dieser Elemente ein sogenanntes 'Property-Binding' und auf Ereignisse ein sogenanntes 'Event-Binding' erfolgen. Diese Datenbindungen können mittels webRAD konfiguriert werden, und es lässt sich damit ein rudimentärer Kontrollfluss innerhalb der Anwendung abbilden - ohne dazu dezidiert Code schreiben zu müssen.

Layoutelemente dienen der Gruppierung von anderen Elementen. Sie sind die einzigen Elemente, die Kind-Elemente innerhalb der Baumstruktur zulassen. In der aktuellen Version von webRAD werden folgende Layoutelemente unterstützt:

- Zeilenlayout
  Gruppiert untergeordnete Elemente vertikal an.

- Spaltenlayout
  Gruppiert untergeordnete Elemente horizontal an.
- Tablayout
  Erstellt für jedes untergeordnete Element ein Tab-Element.
- Kartenlayout
  Gruppiert alle untergeordnete Elemente in ein Karten-Element.

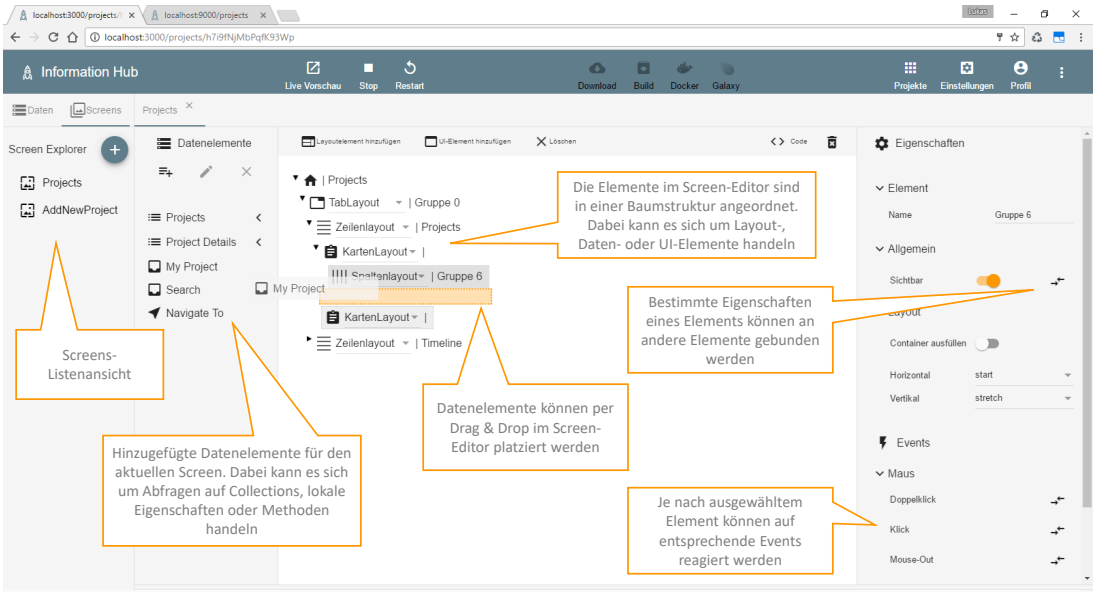

Abbildung 5.5: Screenshot und Illustration des Screen-Editors in webRAD, der zur Erstellung von Screen dient

Bei den Datenelementen handelt es sich um Elemente, welche über eine inhärente Datenbindung zwischen dem Template und der Komponenten-Klasse einer Angular-Komponente verfügen. Je nach Datentyp der Datenbindung stellt webRAD passende UI-Elemente zur Auswahl. Zum Beispiel kann bei einem Boolean-Typ zwischen einer Checkbox und einem Slider oder bei einer Abfrage zwischen einer Tabelle und einer statischen Liste ausgewählt werden. Folgende Datenelemente können in einem Screen hinzugefügt werden:

- Lokale Eigenschaft
  Mittels lokaler Eigenschaften können Werte in einer lokalen Variable auf dem Client gespeichert werden. Die lokale Eigenschaft besitzt einen Datentyp, welcher beim Hinzufügen einer lokalen Eigenschaft angegeben werden muss. Je nach Datentyp können adäquate UI-Elemente innerhalb des Screen-Editors ausgewählt werden.
- Abfrage
  Mithilfe von Abfragen können Daten aus Collections angezeigt, hinzugefügt, gelöscht oder aktualisiert werden. Damit eine Abfrage hinzugefügt werden kann, muss vorher das entsprechende Datenschema im Daten-Editor definiert worden sein. Abfragen können, wie in Abbildung 5.6 ersichtlich, geändert werden, indem Filter- oder Sortierungskriterien hinzugefügt werden.
- Methode
  Mittels Methoden kann auf Benutzerinteraktionen reagiert und somit ein Kontrollfluss innerhalb der Anwendung abgebildet werden. Derzeit stellt webRAD lediglich eine Navigationsmethode zur Verfügung, mit deren Hilfe zwischen verschiedenen Screens gewechselt werden kann. Prinzipiell

wurde jedoch bei der Entwicklung von webRAD darauf geachtet, dass für diesen Aspekt weitere vordefinierte oder benutzerdefinierte Methoden in späteren Versionen hinzugefügt werden können.

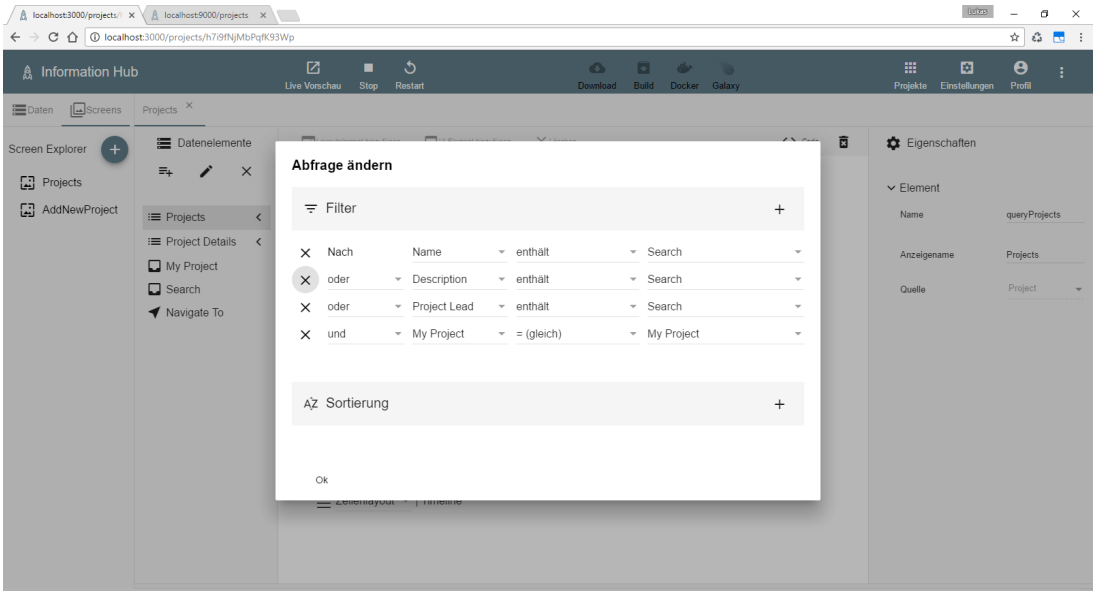

Abbildung 5.6: Screenshot des Dialoges zur Verwaltung von Filter- und Sortierungskriterien von Abfragen in webRAD

UI-Elemente bezeichnen in webRAD Elemente ohne initiale Datenbindung. Nachfolgend werden die aktuell vorhandenen UI-Elemente aufgelistet:

- Icon
  Icons ermöglichen die statische Darstellung von Material Icons[3]. Dabei handelt es sich um eine Sammlung von über 900 Icons, welche in unterschiedlichen Größen und als Web-Font verfügbar sind. Material Icons können in webRAD einfach mittels deren Icon-Name hinzugefügt werden. Diese Eigenschaft wird auch als 'Ligatures' bezeichnet und erhöht wesentlich die Lesbarkeit des HTML-Codes. Ligatures werden automatisch vom Browser durch die numerische Repräsentation ersetzt. Es ist daher erforderlich, dass der eingesetzte Browser dieses Feature nativ unterstützt, was bei allen modernen Browsern der Fall ist.

- Button
  Buttons zählen zu den fundamentalen Elementen der Benutzerinteraktion. Aktuell unterstützt webRAD fünf verschiedene Arten von Buttons, welche sich in Form und Aussehen unterscheiden.

- Bild
  Mittels diesem UI-Element kann ein statisches Bild innerhalb der Anwendung dargestellt werden. In der prototypischen Implementierung von webRAD ist noch keine ausgearbeitete Bildverwaltung inklusive Bild-Upload vorgesehen. Es kann daher aktuell lediglich eine Bild-URL angegeben werden.

Abbildung 5.7 zeigt abschließend einen Ausschnitt einer beispielhaften Anwendung, welche mittels webRAD erstellt wurde. Hervorzuheben ist dabei, dass es sich um eine funktionsfähige, reaktive Webapplikation handelt. Die angezeigten Daten können gefiltert und geändert werden. Aufgrund der reaktiven Eigenschaft einer Meteor-Anwendung wird jede Datenänderung in nahezu Echtzeit mit allen aktiven Clients synchronisiert. Zusätzlich kann zwischen unterschiedlichen Screens navigiert werden, wobei

[3]Material Icons - https://material.io/icons/

sich aufgrund des Einsatzes von Angular Material jeder Screen weitgehend automatisch an die Größe des jeweiligen Bildschirms anpasst. Eine mittels webRAD generierte Anwendung kann daher zumindest in einem großen Umfang als 'responsive' bezeichnet werden.

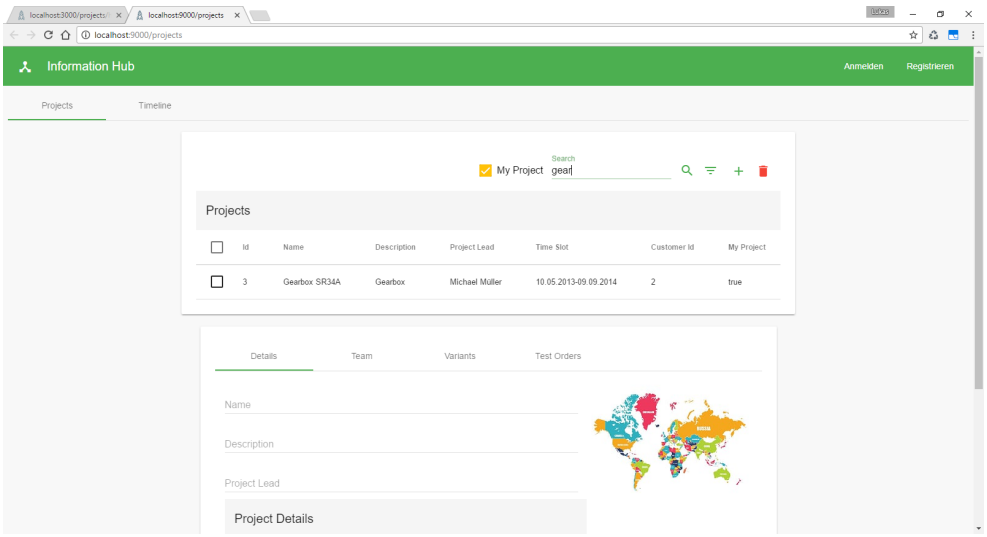

Abbildung 5.7: Screenshot der Live-View einer beispielhaften Anwendung, welche mittels webRAD erstellt wurde

5.1.5 Projekteinstellungen

In den Projekteinstellungen können, wie Abbildung 5.8 veranschaulicht, allgemeine Einstellungen wie der Titel oder das Icon eines Projektes angegeben werden. Mittels der Startscreen-Eigenschaft kann ein vorher erstellter Screen ausgewählt werden, welcher initial beim Start der Anwendung geladen werden soll. Wird kein Startscreen deklariert, verwendet webRAD automatisch einen Standard-Startscreen. Zusätzlich kann in den Projekteinstellungen angegeben werden, ob eine Benutzeranmeldung für die Anwendung gewünscht ist oder nicht. Dabei handelt es sich aktuell um eine einfache Anmeldung mittels Emailadresse und Kennwort eines Benutzers - ohne erweiterte Rechteverwaltung. Schließlich können innerhalb der Projekteinstellungen die elementaren Designfarben (Primär-, Akzent- und Warnfarbe) der Anwendung selektiert werden.

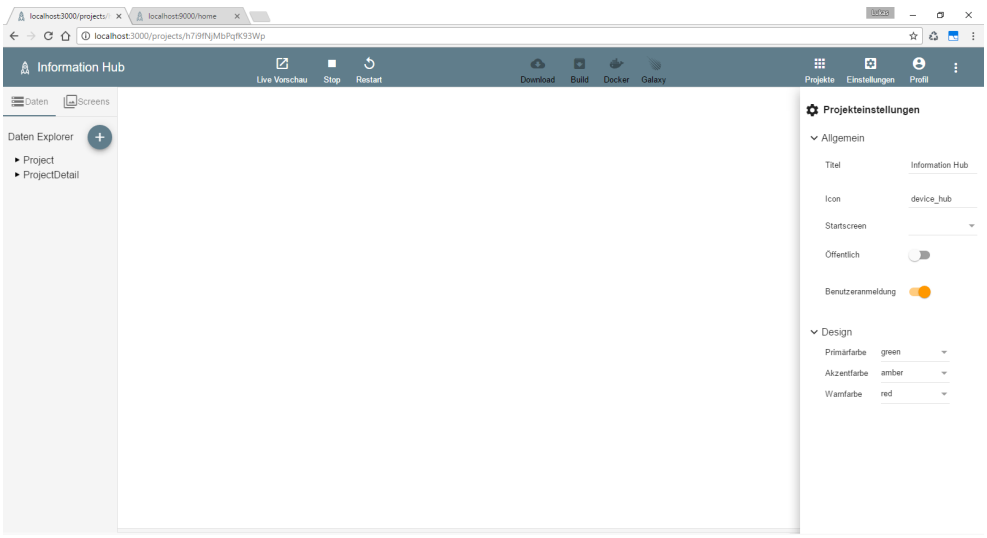

Abbildung 5.8: Screenshot der Projekteinstellungen in webRAD

5.1.6 Code-Download und Deployment

Eine essentielle Eigenschaft von webRAD zur Verwirklichung eines kontinuierlichen Entwicklungsprozesses besteht darin, dass der von webRAD generierte Code heruntergeladen werden kann. Dazu reicht ein Klick auf den Button 'Download' und das gesamte Projekt wird komprimiert und steht danach zum Download bereit. Damit Benutzer in der Zwischenzeit nicht auf den Abschluss des Prozesses warten müssen, werden sie per Email benachrichtigt und können über einen Link die komprimierte Datei herunterladen. Der heruntergeladene Code beinhaltet ein reines Meteor/Angular-Projekt ohne Abhängigkeiten zu webRAD.

Der Editor webRAD unterstützt auch die direkte Bereitstellung als reine Node-Applikation. Dazu muss lediglich, wie in Abbildung 5.9 dargestellt, eine entsprechende Serverarchitektur ausgewählt werden. Derzeit können Meteor-Anwendung auf folgende Plattformen kompiliert werden:

- Windows 32bit
- Linux 32bit
- Linux 64bit
- OSX 64bit

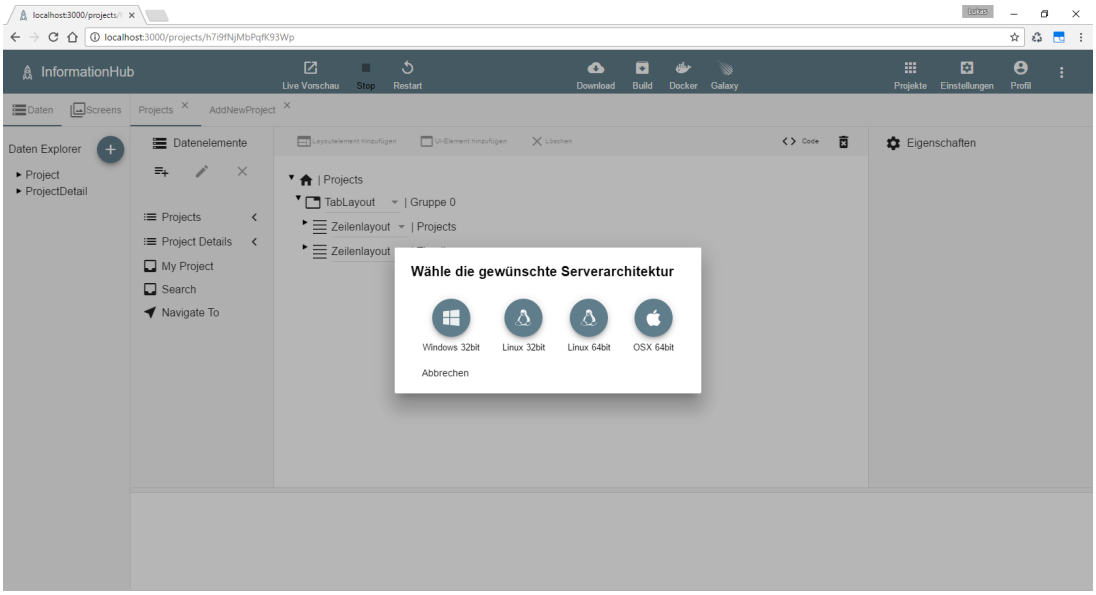

Abbildung 5.9: Screenshot des Dialoges zur Auswahl der gewünschten Serverarchitektur während des Deployment-Prozesses in webRAD

Der Build-Vorgang ist vergleichbar mit dem Code-Download. Benutzer werden ebenfalls per Email und einem Link zum Download des komprimierten Projekts benachrichtigt. Beim generierten Build-Paket handelt es sich um eine reine Node-Applikation, und es besteht weder eine Abhängigkeit zu webRAD noch eine Abhängigkeit zu Meteor. Dadurch kann die Anwendung auf einem herkömmlichen Node-Server bereitgestellt werden. Die einzige Bedingung besteht darin, dass auf dem entsprechenden Server die korrekte Node-Version installiert ist und eine aufrechte Verbindung zu einem MongoDB-Server besteht. Die derzeit eingesetzte Meteor-Version 1.4.x in webRAD erfordert die Node-Version 4.6.2 auf dem Server. Die Anwendung kann anschließend mittels dem node-Befehl und den Argumenten ROOT_URL und MONGO_URL auf dem Server gestartet werden. Dabei entspricht die ROOT_URL der Basis-URL des Meteor-Projekts und die MONGO_URL einer Verbindungszeichenfolge beziehungsweise einer URI zur MongoDB-Datenbank.

5.2 Ausgewählte technische Details

Nachdem sich das Kapitel 5.1 dem Aufbau und der Funktionsweise von webRAD aus Anwendersicht gewidmet hat, werden in diesem Kapitel einige ausgewählte technische Details der Implementierung genauer betrachtet. Zuerst wird die im gesamten Projekt eingesetzte Programmiersprache TypeScript und die Verzeichnisstruktur von Meteor-Anwendungen theoretisch betrachtet. Anschließend wird grob der clientseitige Aufbau von webRAD anhand der Angular-Modulstruktur und die verschiedenen Gruppen von UI-Elementen in webRAD veranschaulicht.

5.2.1 TypeScript

Bei TypeScript handelt es sich um eine relativ neue Programmiersprache, welche von Microsoft und der TypeScript-Community als Open-Source-Projekt entwickelt wurde. Bereits aus dem Namen TypeScript lässt sich der erstrangige Vorteil der Programmiersprache ableiten - die Unterstützung von Typen. Dieser Aspekt ermöglicht JavaScript-Entwicklern die Verwendung von produktiven Entwicklungswerkzeugen und den Einsatz von bewährten Praktiken, wie statische Typenüberprüfung oder Code-Refaktorierung. Typen sind in TypeScript optional und durch die Typen-Inferenz führen bereits einige wenige Annotationen von Typen zu einem wesentlichen Unterschied bei der statischen Überprüfung des Codes. (TypeScript-Team, 2017)

TypeScript unterstützt die neuesten JavaScript-Features, einschließlich der Features in ECMAScript 2015 und zukünftigen Vorschlägen, wie zum Beispiel asynchrone Funktionen und Dekoratoren. Diese Features stehen während der Entwicklung zur Verfügung und werden anschließend in einfaches JavaScript transpiliert, welches je nach Konfiguration auf ECMAScript 3 oder neueren Versionen, basiert. (TypeScript-Team, 2017)

Einer der größten Herausforderungen bei JavaScript besteht darin, den Code weniger fehleranfällig und damit JavaScript geeigneter für umfangreichere Anwendungen zu machen. In objektorientierten Programmiersprachen existieren dafür bewährte Lösungen wie die Modularität oder strikte Typenüberprüfung. Obwohl objektorientierte Prinzipien in eingeschränkter Art und Weise in JavaScript zur Verfügung stehen, stellte sich eine effektive Typenüberprüfung als problematische Aufgabe dar. Es existieren zwar schon seit einigen Jahren verschiedene Lösungen wie zum Beispiel der Closure-Compiler und GWT von Google oder auch eine Reihe von unterschiedlichen C# nach JavaScript Compilern, aber diese erfordern jeweils die Einhaltung einer erheblichen Anzahl an Regeln, damit der entsprechende Compiler effektiv und zuverlässig funktioniert. (Goldshtein, 2016)

Bei der Entwicklung von TypeScript wurde der Fokus auf die Lösung dieses Problems gerichtet. TypeScript dient der Erstellung einer Programmiersprache, welche die Flexibilität von JavaScript möglichst beibehält und gleichzeitig eine optionale und effektive Typenüberprüfung mit minimalen Aufwand für Entwickler garantiert. (Goldshtein, 2016)

Die Typenüberprüfung von TypeScript basiert auf der Form der entsprechenden Typen. Dieser Aspekt wird auch als 'duck typing' bezeichnet und in TypeScript mittels Interfaces aufgegriffen, welche die Form von Typen definieren. (Goldshtein, 2016)

Die Konfiguration von TypeScript wird im Allgemeinen mittels einer dafür vorgesehenen JSON-Datei mit dem Namen 'tsconfig.json' vorgenommen. Eine typische Konfiguration von Typescript innerhalb einer Meteor-Applikation ist in Listing 4.19 im Kapitel 4.14 (Integration von Angular in Meteor) ersichtlich.

5.2.2 Meteor Verzeichnisstruktur

Standardmäßig werden alle JavaScript-Dateien innerhalb eines Meteor-Projektverzeichnisses gebündelt und sowohl auf dem Server als auch dem Client geladen. Dieser Umstand ist in den meisten Fällen nicht gewünscht und Meteor berücksichtigt daher die Bezeichnung der Dateien und Verzeichnisse innerhalb eines Projektes. Die Bezeichnung beeinflusst daher die Reihenfolge und die Umgebung in der die jeweiligen Dateien geladen werden. Nachfolgend werden die wichtigsten Verzeichnisnamen aufgelistet, welche von Meteor speziell behandelt werden:

- imports
  Dateien innerhalb dieses Verzeichnisses werden in keiner Umgebung vorab geladen, sondern müssen mittels des EcmaScript2015-Modul-Systems und deren imports-Syntax importiert werden. Dieser Vorgang wird auch als 'lazy-loading' bezeichnet.

- node_modules
  Verzeichnisse mit dieser Bezeichnung werden nicht geladen. In das node_module werden Node-Pakete installiert, welche in der Anwendung mittels der imports-Syntax importiert werden können.

- client
  Alle Dateien innerhalb dieses Verzeichnisses werden in der Server-Umgebung nicht geladen. HTML-Dateien werden innerhalb von Meteor anders als in vergleichbaren serverseitigen Frameworks behandelt. Zuerst durchsucht Meteor alle HTML-Dateien nach den drei Elementen: <head>, <body> und <template>. Anschließend werden die head- und body-Bereiche separat in einen jeweils einzigen Bereich zusammengefügt, welcher beim initialen Laden der Anwendung an den Client übermittelt wird. Damit wird die Wartezeit für Benutzer beim Laden der Anwendung signifikant reduziert.

- server
  Dieses Verzeichnis wird nicht an den Client übermittelt und kann daher in der Client-Umgebung nicht geladen werden. Sensibler Code, wie zum Beispiel Code, der Kennwörter oder einen Mechanismus zur Authentifizierung beinhaltet, sollte in diesem Verzeichnis abgelegt werden, damit er nicht auf dem Client verfügbar ist.

- public
  Jegliche Dateien innerhalb dieses Verzeichnisses werden unverändert dem Client zur Verfügung gestellt. Das public-Verzeichnis ist der ideale Ort für favicon.ico, robots.txt oder ähnliche Dateien.

- private
  Alle Dateien innerhalb eines Verzeichnisses mit der Bezeichnung private sind nur über den Server-Code zugreifbar und können mittels der Assets-API geladen werden. Ein solches Verzeichnis kann für private Dateien eingesetzt werden, bei denen nicht gewünscht ist, dass sie außerhalb des Projektes zugänglich sind.

- tests
  Jedes Verzeichnis mit dieser Bezeichnung wird in keiner Umgebung geladen. Ein solches Verzeichnis ist für Test-Code vorgesehen, welcher außerhalb von Meteors integrierten Test-Tools ausgeführt werden soll.

- .meteor, .git, packages, programs
  Diese Verzeichnisse werden als Teil des Meteor-Applikations-Codes nicht geladen.

(Goldshtein, 2016)

5.2.3 Angular-Modulsystem

Die clientseitige Implementation von webRAD baut auf Angular auf und verwendet dessen integriertes Modulsystem. Module in Angular helfen gemeinsamen Code und Code mit ähnlicher Funktionalität logisch zusammenzufassen. Abbildung 5.10 illustriert alle Module innerhalb von webRAD einschließlich deren Abhängigkeiten. Jede Anwendung benötigt zumindest ein Root-Modul, welches konventionell als AppModule bezeichnet wird. Bei den Modulen Auth, ProjectList, Project und Screen-Designer handelt es sich um sogenannte Feature-Module, weil sie zusammengehörende Bereiche in der Anwendung widerspiegeln. Das Shared-Modul ist ein Hilfsmodul, welches gemeinsamen Code beinhaltet, der in anderen Modulen wiederverwendet werden kann. Die restlichen Module werden entweder von Angular bereitgestellt, wie beispielsweise das Browser- oder Router-Modul oder stellen extern importierte Module dar.

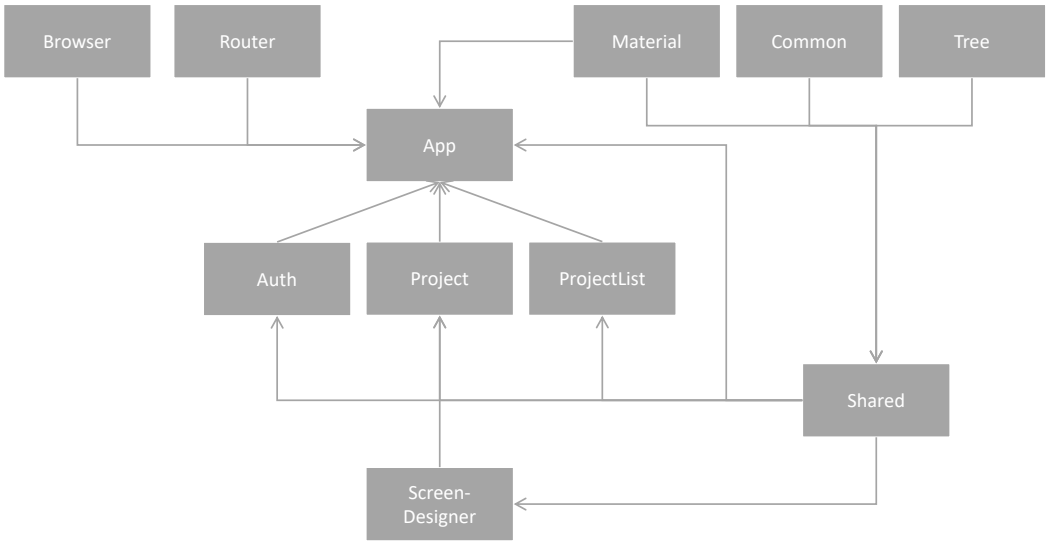

Abbildung 5.10: Angular-Module innerhalb von webRAD einschließlich deren Abhängigkeiten

5.2.4 UI-Elemente

In webRAD existieren fünf verschiedene Typen von UI-Elementen mit unterschiedlichen Funktionalitäten, welche als abstrakte Klassen implementiert wurden und die alle von einer ebenfalls abstrakten Basisklasse erben. Das Diagramm in Abbildung 5.11 stellt dazu schematisch den Aufbau und die Klassenhierarchie der unterschiedlichen UI-Elemente in webRAD dar. Im Allgemeinen können dabei zwischen folgenden Typen unterschieden werden:

- SingleDataControl
  Diese Art von UI-Elemente können genau einen Datenwert mit einem spezifischen Datentyp darstellen beziehungsweise an einen solchen Datenwert gebunden werden. Aus diesem Grund muss bei einer konkreten Implementierung dieser abstrakten Klasse zwingend ein Datentyp angegeben werden, welcher das entsprechende Element unterstützt. Aktuell stellt webRAD folgende SingleDataControl-Elemente bereit: TextBox, LabelControl, CheckBox und SlideToggle.

- MultiDataControl
  MultiDataControl-Elemente wie die DataTable, List oder Combobox unterstützen die Darstellung von mehreren Datenwerten.

- LayoutControl
  Bei diesen Elementen handelt es sich um Layout-Elemente, welche Kind-Elemente beinhalten können. Aus diesem Grund müssen in den konkreten Implementierungen zusätzlich die Methoden 'generateStartTag', 'generateEndTag', 'generateChildStartTag' und 'generateChildEndTag' überschrieben werden. Derzeit stellt webRAD folgende Layout-Elemente zur Verfügung: RowLayout, ColumnLayout, CardLayout und TabLayout.

- NoDataControl
  Darunter werden in webRAD Elemente wie das Image- oder das Icon-Element verstanden, welche über keine inhärente Datenbindung verfügen. Eine Datenbindung zu Attributen der Elemente ist jedoch sehr wohl möglich.

- CommandControl
  CommandControl-Elemente können in webRAD an Methoden anstelle von Datenwerten gebunden werden. Prominenter Vertreter dieser Gruppe ist das Button-Element.

Abbildung 5.11: Klassendiagramm der verschiedenen UI-Elemente in webRAD

Konkrete Implementierungen von UI-Elementen müssen von einer dieser aufgeführten abstrakten Klassen ableiten und die Methoden 'generateCustomHTML' und 'generateCustomCss' implementieren. Listing 5.1 veranschaulicht eine konkrete Implementierung beispielhaft anhand des CheckBox-Elementes. Zuerst werden im Konstruktor zwei Attribute und ein Icon definiert, welche dem Basiskonstruktor übergeben werden können. Zusammen mit dem ebenfalls übergebenen Datentyp verfügt webRAD über ausreichend Informationen, damit das Element innerhalb des Editors dargestellt und für korrelierende Datentypen auswählbar wird. Die Methode 'generateCustomHTML' ist für die Angabe der HTML-Template-Syntax des Elementes vorgesehen, welche zeilenweise der 'html'-Eigenschaft der Basisklasse hinzugefügt werden kann. Auf ähnliche Art und Weise können innerhalb der 'generateCustomCss' benutzerdefinierte CSS-Anweisungen für das entsprechende Element angegeben werden.

```
export class CheckBoxControl extends SingleDataControl {
    constructor() {
        let align: Attribute = {
            type: AttributeTypeEnum.property,
            attributeGroup: AttributeGroupEnum.appearance,
            name: 'align',
            displayName: 'Label Ausrichtung',
            dataType: DataTypeEnum.String,
            bindable: false,
            value: 'start',
            choices: [{ displayValue: 'links', value: 'start' }, {
               displayValue: 'rechts', value: 'end' }]
        };
        let disabled: Attribute = {
            type: AttributeTypeEnum.property,
            attributeGroup: AttributeGroupEnum.general,
            name: 'disabled',
            displayName: 'Deaktiviert',
            dataType: DataTypeEnum.Boolean,
            bindable: true,
            value: false
        };
        let icon: Icon = { ligature: 'checkbox', source: IconSourceEnum.
           selfHosted };

        super(ControlEnum.CheckBox, 'CheckBox', DataTypeEnum.Boolean,
           icon, [align, disabled]);
    }

    generateCustomHtml(screenElement: ScreenElement) {
        this.html.push(`<md-checkbox ${this.getDataBindingBlock(
           screenElement)} ${this.getAttributeBlock(screenElement)} ${
           this.getEventBlock(screenElement)}>`);
        this.html.push('\t' + `${this.getDisplayName(screenElement)}`);
        this.html.push('</md-checkbox>');
    }

    generateCustomCss() {
        this.addCssBlock('md-checkbox', [{ name: 'margin', value: this.
           defaultMargin }]);
    }
}
```

Listing 5.1: Konkrete Implementierung eines UI-Elementes am Beispiel des CheckBox-Elementes

Abbildung 5.12 veranschaulicht das Ergebnis, des in Listing 5.1 definierten CheckBox-Elementes. Das UI-Element, in diesem Fall eine CheckBox, kann sofern der Datentyp des zugrundeliegenden Datenwerts übereinstimmt, im Baum des Screen-Editors ausgewählt werden. Dabei wird das definierte Icon und der Anzeigename, welche beim Aufruf des Basiskonstruktors angegeben wurden, dargestellt. Die ebenfalls im Konstruktor übergebenen Attribute und Events werden auf der rechten Seite des Screen-Editors angezeigt. Dabei ist zu berücksichtigen, dass Standard-Attribute beziehungsweise Standard-Events zu jedem Element automatisch hinzugefügt werden. Hervorzuheben ist, dass je nach angegebenem Datentyp beim Attribut automatisch ein entsprechendes Eingabe-Element verwendet wird. Attribute, welche an andere Datenwerte gebunden werden können, werden mit einem Button gekennzeichnet, mit dessen Hilfe eine Datenbindung vorgenommen werden kann.

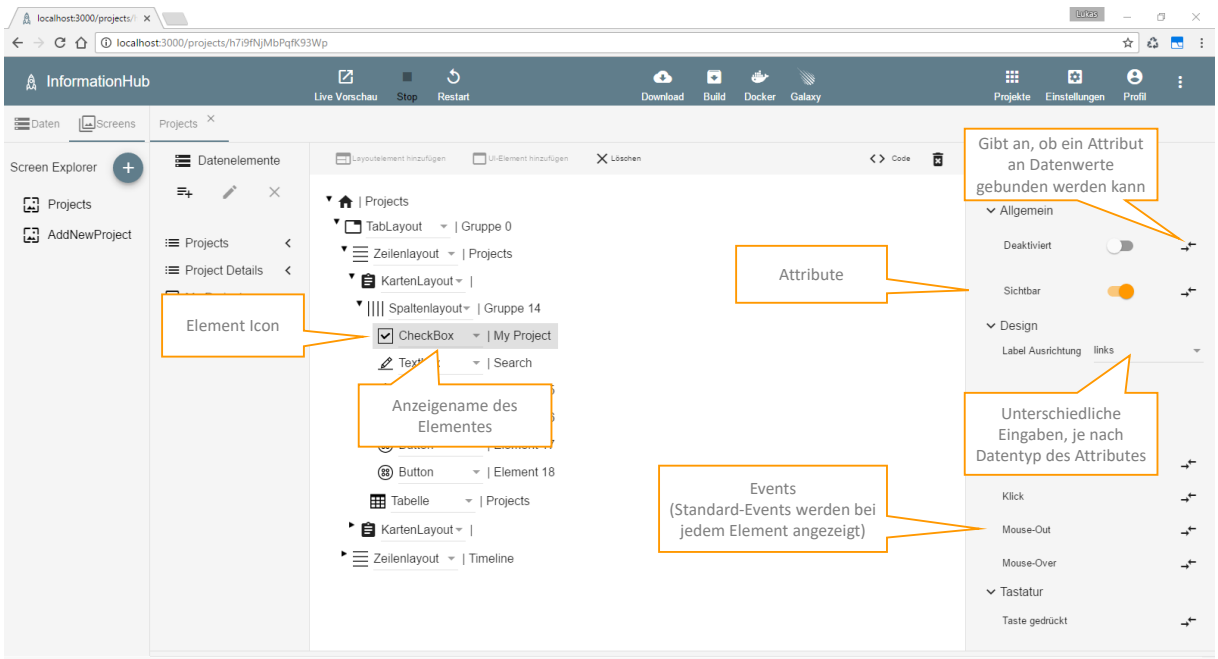

Abbildung 5.12: Screenshot des Resultats der Implementierung des CheckBox-Elements

5.3 Herausforderungen

Bereits während der Konzeption für die prototypische Implementierung war absehbar, dass die eingesetzten Frameworks beziehungsweise Technologien eine spezifische Herausforderung mit sich bringen, weil ein überwiegender Teil davon sich zu diesem Zeitpunkt noch in der Entwicklungsphase befand. Ein Mehraufwand aufgrund von erwartbaren strukturellen Änderungen in den jeweiligen Technologien, welche eine Refaktorierung von Teilen oder des gesamten Codes zur Folge hatte, wurde dennoch akzeptiert. Einerseits bestand eine der zentralen Randbedingung der Arbeit darin, möglichst aktuelle Technologien einzusetzen. Andererseits bringen genau diese neuen Technologien essentielle Vorteile mit sich, auf die in weiterer Folge in der Implementierung nicht verzichtet werden wollte. Beispielhaft sei an dieser Stelle die verbesserte Modularisierung, Performance und die nahtlose Integration von TypeScript mit all seinen Vorteilen in Angular 2 im Vergleich zu Angular 1 erwähnt.

Eine weitere Problematik bestand in der Wahl und Implementierung eines geeigneten Screen-Editors, welcher eine optimale Benutzererfahrung für End-User bereitstellt. Die naheliegende Variante eines What You See Is What You Get (WYSIWYG)-Editors hat sich aufgrund der begrenzten Ressourcen und dem unverhältnismäßig hohen Aufwand für eine prototypische Implementierung als ungeeignet er-

wiesen. Es wurde daher nach einer anderen Möglichkeit gesucht und schlussendlich ein Editor gewählt, welcher in Kapitel 5.1.4 beschrieben wird und für die Konstruktion der Screens eine Baumstruktur verwendet.

Schließlich sind die Schwierigkeiten, welche die Umsetzung einer funktionierenden Live-Vorschau mit sich bringen, zu erwähnen. Innerhalb eines laufenden Meteor-Prozesses auf dem Server können keine weiteren Instanzen parallel gestartet werden. Es war daher erforderlich, für entsprechende Live-Vorschauen eine eigene Node-Instanz als überwiegend unabhängigen Kind-Prozess zu starten. Daraus resultierend war die Implementierung einer übergeordneten Session-, Port- und Prozessverwaltung notwendig, damit die jeweiligen Prozesse gestartet beziehungsweise wieder ordnungsgemäß beendet und die entsprechenden Server-Ressourcen wieder freigegeben werden.

Kapitel 6

Zusammenfassung und Ausblick

> *“ It’s very easy to predict the future. People do it all the time. What you can’t do, is get it right. ”*
>
> [Don Norman, The Front Desk, BBC Video, 1995.]

Wie Don Norman im einleitenden Zitat unbestreitbar formuliert, sind Aussagen über die Zukunft immer mit Vorsicht zu betrachten. Nichtsdestotrotz wird in diesem Kapitel versucht, nach einer Zusammenfassung der Arbeit, allgemeine Trends im Bereich der Webentwicklung aufzugreifen und Ideen für zukünftige Arbeiten vorzuschlagen.

Die vorliegende wissenschaftliche Arbeit hat sich anfänglich der Problematik bei der Entwicklung von Software-Demonstratoren angenommen. Im Speziellen wurden dabei relevante Themen aus dem Umfeld eines forschungsnahen Kompetenzzentrums in der Automobilindustrie genauer beleuchtet. Aufgrund der umfangreichen Problemstellung war eine gesamtheitliche Betrachtung verschiedener Teilbereiche der Softwareentwicklung erforderlich, damit ein geeigneter Lösungsansatz aufgezeigt werden konnte. Aus diesem Grund wurde zuerst das Gebiet der End-User-Softwareentwicklung und deren Abgrenzung zu End-User-Programmierung betrachtet. Anschließend wurden ausgewählte Softwarearchitekturen und mögliche Technologien beziehungsweise Frameworks detailliert analysiert. Anhand dieser Erkenntnisse konnte ein für das Kompetenzzentrum - Das virtuelle Fahrzeug Forschungsgesellschaft mbH (ViF) plausibles Lösungskonzept, welches auf den beiden Frameworks Meteor und Angular aufbaut, erarbeitet werden. Aus diesem Vorschlag heraus galt es, die beiden Technologien in zwei Kapiteln zur Einführung von Meteor beziehungsweise Angular genauer zu erörtern.

Nach dieser theoretischen Abhandlung widmet sich das Kapitel 5 der praktischen Umsetzung des erarbeiteten Konzepts. Das ausgewählte Framework Meteor stellt Benutzern lediglich eine Kommandozeile als Interaktionsschnittstelle bereit. Damit speziell End-User in den Entwicklungsprozess miteinbezogen werden können, ist ein visueller Editor und eine Bibliothek an Drag & Drop-Elementen erforderlich. Im Rahmen dieser Arbeit wurde dazu eine prototypische Implementierung eines webbasierten Editors mit dem Arbeitstitel ’webRAD’ geschaffen, welcher die Erstellung von reaktiven Webapplikationen, basierend auf Meteor und Angular, ermöglicht. Den Aufbau, die Funktionsweise und einige ausgewählte technischen Details von webRAD wurden in diesem letzten Kapitel ausführlich beschrieben.

Zusammenfassend ist die Entwicklung von webbasierten Anwendungen - und dazu zählen speziell auch die Software-Demonstratoren innerhalb von Forschungsprojekten beim ViF - meist sehr komplex und erfordert eine umfassende Erfahrung in diesem Bereich. Dieser Aspekt führt oft dazu, dass sich die Entwicklung auf relativ wenige Personen beschränkt und für End-User andere Instrumente gefunden werden müssen, welche jedoch Technologiebrüche innerhalb des Entwicklungsprozesses verursachen.

Das in dieser Arbeit vorgestellte Framework Meteor versucht, die Erstellung von qualitativ hochwertigen, webbasierten Anwendungen so einfach wie möglich zu gestalten und damit eine breite Gruppe an Personen anzusprechen. Derzeit ist jedoch die Benutzerinteraktion mit Meteor beschränkt und nicht für End-User geeignet. Genau diese Lücke versucht webRAD zu schließen, indem End-Usern - in der vorliegenden Arbeit sind damit speziell Fachexperten in der Automobilindustrie gemeint - mithilfe eines webbasierten, benutzerfreundlichen Editors die Einbindung in den Entwicklungsprozess erleichtert wird.

6.1 Allgemeine Trends

Die in diesem Kapitel angeführten Trends im Bereich der Webentwicklung basieren auf der persönlichen Meinung des Autors und stützen sich weder auf wissenschaftliche Daten noch auf Experten aus diesem Fachgebiet.

Viele End-User sind es inzwischen gewöhnt, immer komplexere und aufwendigere Webanwendungen zu benutzen. Dabei handelt es sich oft um Echtzeit-Anwendungen, wie zum Beispiel Facebook, Google Docs oder ähnliche Anwendungen von prominenten Unternehmen in der Softwareindustrie, welche sich kaum mehr von üblichen Desktopanwendungen unterscheiden. Aufgrund der Erfahrung, die End-User mit solchen Webanwendungen in ihrer Freizeit sammeln, wird sich dieser Trend vermutlich zunehmend auch in der Arbeitswelt fortsetzen. Anwender werden vermehrt Funktionalitäten in webbasierten Geschäftsanwendungen fordern, welche sie von reaktiven Webanwendungen kennen und schätzen gelernt haben. Dieser Aspekt, dass private Anwender ihre eigene IT-Erfahrung in die Unternehmenswelt einbringen wollen, ist auch unter dem Begriff 'Consumerization of IT' populär geworden und ist in vielen Unternehmen bereits Realität.

Die Bedeutung von JavaScript in der Webentwicklung wird allem Anschein nach, sowohl client- als auch serverseitig, signifikant zunehmen. Technologien wie Node oder TypeScript unterstützen diesen Trend und vereinfachen die Entwicklung von Webanwendungen, weil Kompetenzen und Erfahrungen in einer Programmiersprache gesammelt und auf der Client- und Server-Umgebung eingesetzt werden können.

Schließlich sei an dieser Stelle noch der Trend der Modularisierung in der Webentwicklung kurz angeführt. Microservices als Softwarearchitektur wird immer populärer und auch Frameworks wie Angular oder React unterstützen die Entwicklung von Komponenten mit möglichst loser Kopplung, die nicht eng aneinandergebunden sind. Dieser Aspekt ermöglicht hoch skalierbare und robuste Webanwendungen und wird daher auch in Zukunft eine wichtige Rolle einnehmen.

6.2 Ideen für zukünftige Arbeiten

Der webbasierte Editor zur Erstellung von reaktiven Webapplikationen, welcher in Kapitel 5 vorgestellt wurde, stellt lediglich eine prototypische Implementierung dar. Er soll zur Veranschaulichung der in dieser Arbeit propagierten Konzeptidee für einen kontinuierlichen Prozess bei der Entwicklung von Software-Demonstratoren, speziell bei Forschungsprojekten beim ViF, dienen. Für einen produktiven Einsatz sind weiterführende Implementierungen an webRAD notwendig. Einige Ideen für zukünftige Arbeiten werden daher in diesem Kapitel angeführt.

Die Gestaltung des Kontrollflusses der Anwendung beschränkt sich aktuell für End-User im Wesentlichen auf das Reagieren auf Ereignisse und die Erstellung von statischen Datenbindungen. Hier wäre die Schaffung von mehr Flexibilität wünschenswert, indem zum Beispiel benutzerdefinierter Code bei spezifischen Ereignissen in der Anwendung eingefügt werden könnte. Solche Ereignisse, welche auch

als Hooks bezeichnet werden, könnten serverseitig das Hinzufügen, Löschen oder Aktualisieren von Daten in der Datenbank oder clientseitig das Initialisieren oder Beenden einer Angular-Komponente sein. In diesem Zusammenhang wäre die Integration eines Code-Editors in webRAD erstrebenswert, damit der benutzerdefinierte Code nahtlos und einfach in der selben Umgebung eingefügt werden könnte.

Aktuell verfügt webRAD lediglich über einen rudimentären Daten-Editor für die Modellierung der Anwendungsdaten durch End-User. Die Abbildung von relationalen Daten ist in der derzeitigen prototypischen Implementierung nur mittels manueller Erstellung von Fremdschlüsseln möglich. Dieser Vorgang ist speziell für End-User keine befriedigende Lösung und sollte daher bei einer Weiterentwicklung von webRAD berücksichtigt werden.

Damit komplexere Software-Demonstratoren innerhalb von webRAD erstellt werden könnten, sind in zukünftigen Arbeiten benutzerdefinierte Elemente angedacht. Derzeit kann dazu das Projekt heruntergeladen werden und als herkömmliches Meteor-Angular-Projekt weiterverwendet werden. Benutzerdefinierte Elemente würden es erlauben, diesen Vorgang zu einem späteren Zeitpunkt in einem Forschungsprojekt durchzuführen. Damit könnten End-User länger effektiv am Entwicklungsprozess teilnehmen.

Mit dem Meteor Apollo Projekt[1], welches sich zum Startzeitpunkt dieser Arbeit noch in der Entwicklung befand, unternimmt Meteor den Versuch, abseits von MongoDB andere Datenquellen mittels GraphQL[2] zu integrieren. Das scheint ein viel versprechender Ansatz für die Einbindung unterschiedliche Datenquellen in Meteor zu sein, der in einer weiterführenden Arbeit noch zusätzlich untersucht werden könnte.

Abschließend könnten zukünftig in webRAD weitere Deployment-Möglichkeiten geschaffen werden. Dabei bietet sich insbesondere die von Meteor bereitgestellte Hosting-Plattform Galaxy[3] an, welche speziell für Meteor-Anwendungen konfiguriert ist und spezifische Unterstützung für solche Anwendungen anbietet. Eine weitere Möglichkeit besteht in einer Docker[4]-Integration, indem Docker-Container direkt in webRAD erstellt werden. Damit könnten die in webRAD generierten Anwendungen mühelos auf jede Infrastruktur verteilt werden.

[1] Meteor Apollo - http://www.apollodata.com/
[2] GraphQL - http://graphql.org/
[3] Meteor Galaxy - Hosting-Plattform - https://www.meteor.com/galaxy/
[4] Docker - https://www.docker.com/

Literaturverzeichnis

Angular-Team (2016). *Angular Documentation*. Google. 2016. https://angular.io/docs/ts/latest/ (besucht am 21. 01. 2017) (siehe Seiten 52–67, 69–72).

Bainomugisha, Engineer, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx und Wolfgang de Meuter (2013). "A Survey on Reactive Programming". In: *ACM Computing Surveys (CSUR)* 45.4 (Aug. 2013), Seiten 1–34. ISSN 0360-0300. doi:10.1145/2501654.2501666 (siehe Seiten 35, 36).

Bédard, Jean René (2015). *Isomorphic JavaScript*. 2015. http://isomorphic.net/javascript (besucht am 11. 03. 2017) (siehe Seite 30).

Brad Green, Uri Goldshtein (2016). *Angular 2 + Meteor: The Javascript stack of the future*. 20. Apr. 2016. http://angularjs.blogspot.co.at/2016/04/please-welcome-our-friend-uri.html (besucht am 04. 02. 2017) (siehe Seite 74).

Brehm, Spike (2013). *The future of web apps is - ready? - isomorphic JavaScript*. 8. Nov. 2013. http://venturebeat.com/2013/11/08/the-future-of-web-apps-is-ready-isomorphic-javascript/ (besucht am 24. 09. 2016) (siehe Seiten 30–32).

Burnett, Margaret M. und Brad A. Myers (2014). "Future of End-user Software Engineering: Beyond the Silos". In: *Proceedings of the on Future of Software Engineering*. FOSE 2014. Hyderabad, India: ACM, 2014, Seiten 201–211. ISBN 978-1-4503-2865-4. doi:10.1145/2593882.2593896 (siehe Seite 10).

Chodorow, Kristina (2013). *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. Herausgegeben von Ann Spencer. 2. Auflage. O'Reilly Media, Inc., Mai 2013. ISBN 978-1-449-34468-9 (siehe Seiten 40, 41).

Coleman, Tom und Dominic Nguyen (2016). *Meteor API Docs*. Meteor. 2016. http://docs.meteor.com/ (besucht am 28. 09. 2016) (siehe Seiten 29, 45).

Conway, Melvin Edward (1968). "How do committees invent". In: *Datamation* 14.4 (Apr. 1968), Seiten 28–31 (siehe Seite 15).

Dijkstra, Edsger (1975). *BrainyQuote.com*. Xplore Inc. 1975. http://www.brainyquote.com/quotes/quotes/e/edsgerdijk204332.html (besucht am 27. 09. 2016) (siehe Seite 29).

Fain, Yakov und Anton Moiseev (2016). *Angular 2 Development with TypeScript*. 1. Auflage. Manning Publications Company, 30. Dez. 2016, Seite 456. ISBN 978-1617293122 (siehe Seite 51).

Ferreira, Vasco (2016). *Functional Reactive Programming for Angular 2 Developers - RxJs and Observables*. 3. Feb. 2016. http://blog.angular-university.io/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/ (besucht am 08. 02. 2017) (siehe Seiten 67, 68).

Fowler, Martin und James Lewis (2015). "Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr?" In: *Objektspektrum* 1 (2015), Seiten 14–20. https://www.sigs-datacom.de/

uploads/tx_dmjournals/fowler_lewis_OTS_Architekturen_15.pdf (besucht am 22. 02. 2017) (siehe Seiten 14, 15).

Gall, John (1977). *Systemantics: How Systems Work and Especially How They Fail*. Quadrangle/New York Times Book Co., 1977. ISBN 9780812906745. https://books.google.at/books?id=8dkaAAAAMAAJ (siehe Seite 1).

Goldshtein, Uri (2016). *Build Realtime Web and Mobile Apps With Angular and Meteor*. 15. Jan. 2016. https://angular-meteor.com/ (besucht am 18. 02. 2017) (siehe Seiten 68, 73–75, 86, 87).

Greif, Sacha (2014). *An Introduction To Latency Compensation*. 27. Dez. 2014. https://www.discovermeteor.com/blog/latency-compensation/ (besucht am 27. 09. 2016) (siehe Seite 34).

Greif, Sacha (2015). *What Goes Where: Making Sense of Meteor's Client/Server Split*. 7. Juli 2015. https://www.discovermeteor.com/blog/what-goes-where/ (besucht am 23. 09. 2016) (siehe Seite 32).

Greussing, Lukas und Markus Zoier (2016). "How to Enable a User-centric, Web-based Co-creation Process to Facilitate Software Demonstrator Development in Automotive Engineering". In: *Quality Assurance in Computer Vision & Digital Eco-Systems. Joint Proceedings of the International Workshop on Quality Assurance in Computer Vision and the International Workshop on Digital Eco-Systems*. (Graz, Austria, 17.–19. Okt. 2016). Herausgegeben von Bernhard Peischl und Iulia Nica. CEUR Workshop Proceedings 1711. 18. Okt. 2016, Seite 32. http://ceur-ws.org/Vol-1711/absDECOSYS1.pdf (siehe Seiten 25, 26).

Haviv, Amos Q. (2014). *MEAN Web Development*. Packt Publishing Ltd, Sep. 2014. ISBN 978-1-78398-328-5 (siehe Seiten 22, 23).

Houdek, Frank (2003). "Requirements Engineering Erfahrungen in Projekten der Automobilindustrie". In: *Softwaretechnik Trends* 23.1 (2003). http://pi.informatik.uni-siegen.de/stt/23_1/01_Fachgruppenberichte/FG216/08_Houdek.pdf (besucht am 08. 02. 2017) (siehe Seite 2).

Kambona, Kennedy, Elisa Gonzalez Boix und Wolfgang De Meuter (2013). "An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications". In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. 3. Montpellier, France: Association for Computing Machinery (ACM), 2013, Seiten 1–9. ISBN 978-1-4503-2041-2. doi:10.1145/2489798.2489802 (siehe Seite 35).

Ko, Andrew J., Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, Susan Wiedenbeck, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance und Henry Lieberman (2011). "The State of the Art in End-user Software Engineering". In: *ACM Computing Surveys* 43.21 (3 Apr. 2011), Seiten 1–44. doi:10.1145/1922649.1922658 (siehe Seiten 11, 13).

Lizcano, David, Fernando Alonso, Javier Soriano und Genoveva Lopez (2013). "A Web-centred Approach to End-user Software Engineering". In: *ACM Transactions on Software Engineering and Methodology* 22.36 (4 Okt. 2013), Seiten 1–29. doi:10.1145/2522920.2522929 (siehe Seiten 11, 12).

Meteor-Team (2016). *Meteor Guide*. Meteor Development Group. 2016. https://guide.meteor.com/ (besucht am 09. 02. 2017) (siehe Seiten 46–49).

Moura, Ana Lúcia De und Roberto Ierusalimschy (2009). "Revisiting Coroutines". In: *ACM Trans. Program. Lang. Syst.* 31.2 (Feb. 2009), Seiten 1–31. doi:10.1145/1462166.1462167 (siehe Seite 40).

Musliu, Nysret, Wolfgang Slany und Johannes Gärtner (2013). "Automated Test Case Generation in End-User Programming". In: *End-User Development. 4th International Symposium, IS-EUD 2013, Copenhagen, Denmark, 2013. Proceedings*. Herausgegeben von Yvonne Dittrich, Margaret Burnett,

Anders Mørch und David Redmiles. Band 7897. Springer Berlin Heidelberg, 10.–13. Juni 2013, Seiten 272–277. ISBN 978-3-642-38705-0. doi:10.1007/978-3-642-38706-7_25 (siehe Seite 12).

Nardi, Bonnie A. (1993). *A Small Matter of Programming: Perspectives on End User Computing*. The MIT Press, Massachusetts Institute of Technology, 1993. ISBN 0-262-14053-5. https://mitpress.mit.edu/books/small-matter-programming (siehe Seite 13).

Newman, Sam (2015). *Building Microservices: Designing Fine-Grained Systems*. Herausgegeben von Mike Loukides und Brian MacDonald. 1. Auflage. O'Reilly Media, Feb. 2015. ISBN 978-1-491-95035-7 (siehe Seite 16).

Pichai, Sundar (2014). *BrainyQuote.com*. Xplore Inc. 2014. https://www.brainyquote.com/quotes/quotes/s/sundarpich672180.html (besucht am 21. 02. 2017) (siehe Seite 51).

ReactiveX-Team (2016). *ReactiveX Documentation*. 2016. http://reactivex.io/rxjs/manual/ (besucht am 11. 02. 2017) (siehe Seite 67).

Requena, Juampy Novillo (2015). *What is an isomorphic application?* 10. Juni 2015. https://www.lullabot.com/articles/what-is-an-isomorphic-application (besucht am 30. 09. 2016) (siehe Seite 33).

Rheingold, Howard (2012). *BrainyQuote.com*. Xplore Inc. 2012. https://www.brainyquote.com/quotes/quotes/h/howardrhei560070.html (besucht am 30. 03. 2017) (siehe Seite 77).

Scaffidi, Christopher, Joel Brandt, Margaret Burnett, Andrew Dove und Brad Myers (2012). "SIG: End-user Programming". In: *CHI '12 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '12. Austin, Texas, USA: ACM, 2012, Seiten 1193–1996. ISBN 978-1-4503-1016-1. doi:10.1145/2212776.2212421 (siehe Seite 14).

Susiripala, Arunoda (2014a). *Meteor DDP Analyzer*. 15. Aug. 2014. https://github.com/arunoda/meteor-ddp-analyzer (besucht am 01. 10. 2016) (siehe Seite 45).

Susiripala, Arunoda (2014b). *Meteor Explained - A Journey Into Meteor's Reactivity*. 6.1. Gumroad, Inc., 24. Sep. 2014. https://gumroad.com/l/meteor-explained (siehe Seiten 33, 34, 38–40, 42–44).

Syromiatnikov, Artem und Danny Weyns (2014). "A Journey through the Land of Model-View-Design Patterns". In: *2014 IEEE/IFIP Conference on Software Architecture*. Institute of Electrical und Electronics Engineers (IEEE), Apr. 2014, Seiten 21–30. doi:10.1109/WICSA.2014.13 (siehe Seiten 16–20).

Tilkov, Stefan und Steve Vinoski (2010). "Node.js: Using JavaScript to Build High-Performance Network Programs". In: *IEEE Internet Computing* 14 (6 1. Nov. 2010), Seiten 80–83. doi:10.1109/mic.2010.145 (siehe Seite 38).

TU Graz, IST (2017). *IST - Institut für Softwaretechnologie, TU Graz*. 2017. http://www.ist.tugraz.at/ (besucht am 28. 02. 2017) (siehe Seite 2).

TypeScript-Team (2017). *TypeScript Documentation*. 2017. https://www.typescriptlang.org/docs/tutorial.html (besucht am 24. 03. 2017) (siehe Seiten 81, 86).

Virtual-Vehicle (2017). *Virtual Vehicle - Kompetenzzentrum - Das virtuelle Fahrzeug, Forschungsgesellschaft mbH*. 2017. http://www.v2c2.at (besucht am 27. 02. 2017) (siehe Seite 1).

Walther, Stephen (2014). *Don't Do, React! Understanding Meteor Reactive Programming*. 5. Dez. 2014. http://stephenwalther.com/archive/2014/12/05/dont-do-react-understanding-meteor-reactive-programming (besucht am 03. 10. 2016) (siehe Seite 37).

Wehrle, Rick (2015). *Seven Core Principles of Meteor*. 9. März 2015. https://de.slideshare.net/wehrlock/20150309-seven-core-principles-of-meteor-45620129 (besucht am 04. 10. 2016) (siehe Seite 30).

Wirth, Niklaus (1997). *BrainyQuote.com*. Xplore Inc. 1997. https://www.brainyquote.com/quotes/quotes/n/niklauswir306074.html (siehe Seite 5).