
# Weiterentwicklung einer desktoporientierten Anwendung zu einer modernen Webapplikation im Bereich der amtlichen Wertermittlung

# Further Development of a Desktop Application into a Modern Web Application in the Field of Official Valuation of Properties

**Bachelorarbeit**

im Studiengang

Geoinformatik Bachelor of Science

von Malte Wrogemann

| | |
|---|---|
| Erstprüfer: | Prof. Dr. Stefan Schöf<br>Jade Hochschule Oldenburg |
| Zweitprüfer: | Dr. Marcel Ziems<br>Landesamt für Geoinformation und Landesvermessung Niedersachsen |
| Ausgabe: | 03. Juni 2019 |
| Abgabe: | 02. August 2019 |
| Vorgelegt von: | Malte Wrogemann |
| Matr.-Nr.: | 6011623 |

JADE HOCHSCHULE
Wilhelmshaven Oldenburg Elsfleth

Name: Wrogemann

Vorname: Malte

Matrikelnummer: 6011623

Erklärung gemäß § 21 (5) Allgemeiner Teil (Teil A) der Prüfungsordnung für die Bachelor-Studiengänge (BPO) an der Jade Hochschule Wilhelmshaven/Oldenburg/Elsfleth in der Fassung der Bekanntmachung vom 08. Dezember 2004 (VkBl. Nr. 37/2004), zuletzt geändert am 21.10.2014 (VkBl. Nr. 56/2014 vom 24. November 2014)

Die Bachelor-Arbeit ist

☒ eine Einzelarbeit.

☐ eine Gruppenarbeit zusammen mit der/dem Studierenden:

_____.

Ich versichere hiermit, die Bachelor-Arbeit

☐ bei einer Gruppenarbeit den/die Teil(e)

_____

selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Oldenburg, 02.08.2019
(Ort, Datum)

Malte Wrogemann
(Unterschrift Studierende/r)

# Abstract

In dieser Arbeit wurde ein Konzept für Online-Formulare erstellt, das die Anforderungen an komplexe Formulare in der Verwaltung erfüllt und modulartig in den Webapplikationen des Landesamtes für Geoinformation und Landesvermessung Niedersachsen (LGLN) implementiert werden kann. Bei der Umsetzung dieses Konzeptes mithilfe von Angular, Docker, gRPC und Go haben sich die Technologien bewährt und für eine weitere Betrachtung durch das LGLN empfohlen. Gerade gRPC für die Nutzung im Web steht zum Zeitpunkt dieser Arbeit erst seit wenigen Monaten zur Verfügung und hat sich neben der Verwendung in Microservices-Architekturen auch als ein interessantes Kommunikationsmittel über die Web-Schnittstelle herausgestellt.

# Gliederung

# Abbildungen

# Abkürzungen

| | |
|---|---|
| API | Application Programming Interface |
| AWS | Amazon Web Services |
| CLI | Command Line Interface |
| CSS | Cascading Style Sheets |
| DOM | Document Object Model |
| GAG | Gutachterausschuss für Grundstückswerte |
| GCP | Google Cloud Plattform |
| GraphQL | Graph Query Language |
| gRPC | Googles Remote Procedure Call |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| LGLN | Landesamt für Geoinformation und Landesvermessung Niedersachsen |
| NPM | Node Package Manager |
| PHP | PHP: Hypertext Preprocessor |
| REST | Representational State Transfer |
| RPC | Remote Procedure Call |
| SASS | Syntactically Awesome Stylesheets |
| SQL | Structured Query Language |
| URI | Uniform Resource Identifier |
| YAML | YAML Ain’t Markup Language / Yet Another Markup Language |

# Glossar

**Application Programming Interface (API)**

Eine API bietet eine Abstraktion für ein Problem und spezifiziert, wie Benutzer mithilfe von Softwarekomponenten, die eine Lösung für das Problem implementieren, reagieren sollen. (vgl. REDDY 2011)

„Eine API spezifiziert die Operationen sowie die Ein- und Ausgaben einer Softwarekomponente. Ihr Hauptzweck besteht darin, eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren, sodass die Implementierung variieren kann, ohne die Benutzer der Softwarekomponente zu beeinträchtigen.“ (BLOCH 2014)

Remote-APIs, wie RESTful APIs und RPC-APIs sind Schnittstellen, die über ein Protocol wie HTTP plattformunabhängig agieren können. (vgl. SPICHALE 2017)

**CSS Präprozessoren**

„CSS-Präprozessoren [wie SASS und LESS] ergänzen CSS mit Variablen, Funktionen und Mixins zu einer runden Sprache und helfen, den Code schlank zu halten.“ (BEZ 2014)

**Command Line Interface (CLI)**

CLIs bieten eine textbasierte Benutzeroberfläche, um mit dem Betriebssystem zu interagieren. Es erlaubt dem Nutzer auf visuelle Rückgaben, über das Eingeben von Befehlen, zu reagieren. (vgl. TECHNOPEDIA 2019a)

**Compiler**

„[Der Compiler ist ein] Systemprogramm, das ein in einer höheren Programmiersprache formuliertes Quellprogramm […] in ein Maschinenprogramm übersetzt.“ (LACKES & SIEPERMANN 2018a)

**Document Object Model (DOM)**

„Mit dem Document Object Model (DOM), einer vom World Wide Web Consortium (W3C) entwickelten Spezifikation für eine Programmierschnittstelle, kann ein Entwickler HTML-Seiten und XML-Dokumente als vollwertige Programmobjekte erstellen und ändern.“ (ROUSE 2014)

**Garbage Collector**

Ein Garbage Collector ist eine Software, die für die automatische Speicherbereinigung zuständig ist. Es gibt Speicher frei, wenn er nicht mehr genutzt wird. (vgl. TECHNOPEDIA 2019b)

**Microservices**

„Microservices sind unabhängig deploybare Module." Sie stehen im Gegensatz zum Deployment-Monolithen. (WOLFF 2018: 3)

**Mobile First**

„Mobile First bezeichnet einen neuen Denkansatz im Webdesign, bei dem die Darstellung auf mobilen Endgeräten die höchste Priorität bei der Webentwicklung einer Website besitzen soll." (ONLINEMARKETING-PRAXIS 2019)

**Open Source**

„[Open Source ist ein] Konzept, nach dem Programme mit ihrem Quellcode ausgeliefert werden. Jeder darf den Quellcode einsehen und verändern. Die Open Source Initiative (OSI) definiert Kriterien, die Open Source Software erfüllen soll." (LACKES & SIEPERMANN 2018b)

**Single Page Application**

„[Die Single Page Application ist eine] Anwendung oder Website, die statt wie üblich auf mehrere Seiten verteilt, komplett auf einer Seite realisiert ist, um den Nutzern den Eindruck und das Gefühl einer Desktop-Anwendung zu vermitteln." (SIEPERMANN 2018)

**Uniform Resource Identifier**

„Im Web gibt es ein einheitliches Konzept für die Vergabe von IDs – nämlich die URI. URIs bilden einen globalen Namensraum, und die Verwendung einer URI als ID stellt sicher, dass Sie die wesentlichen Elemente weltweit eindeutig identifizieren können." (TILKOV et al. 2015)

**Verteilte Systeme**

Ein verteiltes System ist ein System, in dem Hard- und Softwarekomponenten, die sich auf miteinander vernetzten Computern befinden, miteinander kommunizieren und ihre Aktionen koordinieren, indem sie Nachrichten austauschen. (vgl. WERNER 2011)

# 1 Einleitung

Die Softwareentwicklung von Webseiten und Webanwendungen bzw. Webapplikationen hat sich in den letzten Jahrzehnten grundlegend gewandelt. Früher wurden HTML- (Hypertext Markup Language) Seiten auf dem Server meist über Programmiersprachen wie PHP (PHP: Hypertext Preprocessor) gerendert und dann im Browser über das Einbinden von CSS- (Cascading Style Sheets) Dateien gestaltet. JavaScript hatte kaum eine Bedeutung oder wurde nur für kleinere Gimmicks wie zum Beispiel optische Effekte genutzt.

Heute ist die Single Page Application *State of the Art* der modernen Webentwicklung und JavaScript hat sich von einer weitgehend vernachlässigten zu einer der wichtigsten Programmiersprachen in der Web-Branche entwickelt. Die Programmierlogik, die früher fast ausschließlich serverseitig abgelaufen ist, befindet sich heute zu großen Teilen direkt beim Client. Das hat dazu geführt, dass sich in der JavaScript-Umgebung zahlreiche neue Technologien entwickelt haben. In erster Linie sind dabei die drei großen JavaScript-Frameworks Angular, React.js und Vue.js zu nennen. Daneben gibt es heute einige interessante Programmiersprachen wie TypeScript und Elm, die zu JavaScript kompilieren, und der Sprache Typsicherheit und funktionale Programmierparadigmen hinzufügen. Die Popularität von JavaScript geht sogar soweit, dass mit Node.js eine Plattform entwickelt wurde, die eine serverseitige Programmierung mit JavaScript ermöglicht.

Im Backend hat eine Entwicklung zu einer Service-Architektur stattgefunden. Komplexe Programme werden nicht mehr als große, unflexible Monolithen konzipiert, sondern in modulartige, kleine Anwendungen aufgeteilt, die je nach Einsatzgebiet als API- (Application Programming Interface) Services, RESTful- (Representational State Transfer) Services oder Microservices bezeichnet werden. Ein Ziel ist es dabei, besonders aufwändige Arbeiten zu isolieren, um diese horizontal also über mehrere Server hinweg (Cloud Computing) skalieren zu können.

Die Kommunikation verteilter Systeme findet meist über das REST-Programmierparadigma statt. Die lose gekoppelte und wiederverwendbare Schnittstelle eignet sich besonders für öffentliche APIs mit vielen heterogenen Nutzergruppen. Daneben gibt es noch weitere Ansätze wie zum Beispiel das durch Facebook sehr bekannt gewordene GraphQL (Graph Query Language) oder das RPC- (Remote Procedure Call) Verfahren. Mit gRPC hat Google ein Framework entwickelt, das für eine performante Interaktion zwischen Microservices sorgt und derzeit in vielen Diensten innerhalb der Google Cloud Plattform (GCP) zum Einsatz kommt (vgl. SANDOVAL 2018).

Docker ist eine Technologie, die die Entwicklung von Software besonders im Umfeld von Cloud Computing und Microservices revolutioniert hat. Der Standard von Docker für das Virtualisieren von beliebigen Anwendungen in Linux-Containern wird von den großen Cloud-Plattformen unterstützt. GCP, Azure und AWS (Amazon Web Services) nutzen Docker beispielsweise in Zusammenhang mit der Container-Orchestrierungssoftware Kubernetes.

Viele interessante Open Source Softwareprojekte wie Docker und Kubernetes werden derzeit in der relativ neuen Programmiersprache Go entwickelt. Die im März 2012 von Google veröffentlichte, statisch-typisierte und kompilierbare Sprache eignet sich besonders für die Entwicklung von Microservices, CLIs (Command Line Interface) und betriebssystemübergreifender Software (vgl. GOLANG 2019a).

## 1.1 Motivation

Die Entwicklung moderner Webapplikationen besteht heute aus einer großen Zahl an Technologien, die sich ständig weiterentwickeln. Das Landesamt für Geoinformation und Landesvermessung Niedersachsen (LGLN) möchte diese Technologien nutzen, um zukünftige Softwareprojekte moderner zu gestalten. Desktoporientierte Lösungen können nicht mit den universellen Einsatzmöglichkeiten von Webapplikationen mithalten und sollen daher neu realisiert werden.

Eine Aufgabe des LGLN im Bereich der amtlichen Wertermittlung ist die technische Unterstützung der Gutachterausschüsse für Grundstückswerte (GAGs). Diese führen die Kaufpreissammlung und werten diese jährlich in dem Grundstücksmarktbericht aus. Die Informationen erhalten die GAGs aus den Kaufverträgen über Grundstücke, Eigentumswohnungen oder Erbbaurechte direkt von den Notaren (vgl. GAG-NIEDERSACHSEN 2019). Ergänzt werden diese Daten durch Fragebögen, die die Eigentümer verpflichtend ausfüllen müssen. Einer dieser Fragebögen ist als Anlage in dieser Arbeit angehängt (Anlage 1: Fragebogen für Ein- und Zweifamilienhäuser). Dieser ist ein gutes Beispiel für die Komplexität und Fachlichkeit von amtlichen Formularen. Aufgrund der vermeintlich hochsensiblen Käuferdaten kommt es im Fall des GAG-Fragebogens zudem häufig zu Beschwerden.

Die Umsetzung der Formulare im Bereich der amtlichen Wertermittlung des LGLN in Webapplikation soll die Qualität und Quantität der Daten verbessern. Dynamische Formularelemente und eine abgestimmte Programmlogik kann die Nutzererfahrung verbessern und zudem Aspekte wie die Barriere- und Sprachfreiheit mit einbeziehen.

## 1.2 Eingrenzung

Das Ziel dieser Arbeit ist die Aufstellung eines Konzeptes für die systematische Erstellung von Online-Formularen in modernen Webanwendungen und die Umsetzung dieser mit Technologien, die bis jetzt noch nicht beim LGLN verwendet werden.

Die Entwicklung einer ausgefallenen Formularlogik für ein konkretes Projekt und die Begutachtung dessen aus unterschiedlichen Perspektiven wie der Barriere- und Sprachfreiheit ist dagegen kein Ziel dieser Arbeit. Es soll alleine die Grundlage für eine Softwarearchitektur konzipiert werden, die die Entwicklung einer solchen Formularlogik in Zukunft auf eine einheitliche Weise ermöglicht.

Exemplarisch soll das Konzept in einer Webapplikation umgesetzt werden. Dabei wird ein Prototyp als eine Art *Proof of Concept* mit ausgewählten Technologien umgesetzt, welche in dieser Arbeit näher erläutert werden. Dazu zählen im Wesentlichen Angular, Docker, gRPC und Go. Das Projekt stellt die minimalen Grundfunktionen vor, die ein Online-Formular bieten muss.

Alternative Software, Programmiersprachen und Frameworks werden nur am Rande behandelt. Stattdessen stellt diese Arbeit die Einsatzmöglichkeiten der ausgewählten Technologien in den Vordergrund und bewertet diese aus der Perspektive des LGLN. Das Konzept ist in dem Sinne universell, als dass es auch mit anderen Technologien auf eine ähnliche Art und Weise umgesetzt werden könnte.

Weiterhin werden in dieser Arbeit nicht sämtliche Bereiche einer modernen Webapplikation abgedeckt. Es werden weder Konzepte zur Autorisierung und Authentifizierung besprochen, noch wird auf das Logging oder das Monitoring eingegangen. Dementsprechend ist es kein Ziel, eine produktionsbereite Anwendung zu erstellen.

## 1.3 Aufbau

Diese Arbeit ist abgesehen von der Einleitung und der Zusammenfassung in fünf Teile gegliedert. Ausgehend von der Anforderungsanalyse und der Konzepterstellung werden die Grundlagen der Technologien erarbeitet, welche daraufhin für die Umsetzung des Konzeptes genutzt werden. Im Anschluss wird die Lösung bewertet.

In der Anforderungsanalyse (Kap. 2) steht zunächst die Entwicklung der Anforderungen an ein Online-Formular im Vordergrund. Diese findet mithilfe des bereits genannten Fragebogens für Ein- und Zweifamilienhäuser der GAGs statt. Dabei werden Anwendungsfälle aus der Sicht der Nutzer und Entwickler beschrieben und daraus funktionale und nicht-funktionale Anforderungen abgeleitet. Die Anforderungsanalyse ist die Basis der Konzepterstellung.

Das Konzept in Kapitel 3 führt die Anforderungen mit den ausgewählten Technologien Angular, Docker, gRPC und Go zusammen. Es wird ein Konzept entwickelt, aus dem die Aufgaben der unterschiedlichen Komponenten im Frontend und Backend bzw. als Teil der Entwicklungsumgebung hervorgehen. Die Vorstellung der verwendeten Technologien erfolgt dann in Kapitel 4, welches sich mit den Grundlagen beschäftigt.

Die Umsetzung des Konzeptes findet in Kapitel 5 statt. Dabei wird ein gRPC-Protocol, eine Angular Library und ein Go Package entwickelt. Zuletzt wird ein Beispiel erstellt, das zudem weiter auf die Nutzung von Docker eingeht. In Kapitel 6 werden die Ergebnisse evaluiert sowie die Differenzen zur Anforderungsanalyse betrachtet.

# 2 Anforderungsanalyse

Zu Beginn sind alle Anforderungen zu ermitteln, die im Konzept verwirklicht werden sollen. Es geht darum ein System zu erschaffen, dass erweiterbar, wartbar, redundanzfrei, wiederverwendbar und gut verständlich ist (vgl. DUNKEL & HOLITSCHKE 2003: 11). Eine weitere Komponente, die eng mit der Wartbarkeit verknüpft ist, ist die Austauschbarkeit. Diese wird in Softwarearchitekturen z.B. über eine gut dokumentierte Schnittstelle erreicht (vgl. RODEN 2008: 83).

Ausgehend von dem bereits genannten Fragebogen der GAGs werden zunächst Anwendungsfälle beschrieben, die für ein Online-Formular relevant sind. Um ein möglichst ganzheitliches Bild zu erhalten, werden diese aus der Nutzer- und der Entwicklerperspektive betrachtet. Aus den Anwendungsfällen lassen sich dann in Kapitel 2.3 die funktionalen und nicht-funktionalen Anforderungen ermitteln.

Der Fragebogen für Ein- und Zweifamilienhäuser der GAGs ist ein reales Projekt aus der amtlichen Wertermittlung und eignet sich daher als Gegenstand dieser Anforderungsanalyse. Zunächst werden die Fragen und deren Antwortmöglichkeiten, im Folgenden Felder genannt, analysiert, um die unterschiedlichen Feldtypen zu ermitteln und die Formularlogik zu untersuchen.

Besonders Fragebögen, die aus der öffentlichen Verwaltung kommen, sind aufgrund von gesetzlichen Anforderungen oft kompliziert oder in einer sehr fachlichen Sprache formuliert. Diese komplexeren Sachverhalte sollen in einem Online-Formular abgebildet werden können, aber gleichzeitig leichtgewichtiger erscheinen.

## 2.1 Fragebogen

In diesem Abschnitt wird der Fragebogen für Ein- und Zweifamilienhäuser (Anlage 1) untersucht und daraus die wichtigsten Feldtypen sowie die Formularlogik ermittelt. Die Ergebnisse fließen in die Anwendungsfälle in Kapitel 2.2 mit ein.

### 2.1.1 Feldtypen

Für die Qualität von Online-Formularen ist die Wahl der richtigen Feldtypen von entscheidender Bedeutung. Jeder Klick, der bei der Eingabe und Auswahl von Formularfeldern eingespart werden kann, verbessert die Nutzererfahrung. Im Folgenden werden daher die wichtigsten Feldtypen kategorisiert, die zum Mindestumfang eines Formulars gehören.

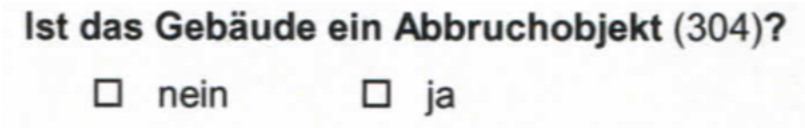


Abbildung 1: Auswahlfelder Typ I (Anlage 1)

Es kann zwischen mindestens zwei Typen von Auswahlfeldern unterschieden werden. Die Abbildung 1 zeigt ein Beispiel, in dem eine klare Entscheidung zwischen einer begrenzten Zahl an Optionen getroffen werden muss. Der zweite Typ (Abb. 2) besteht aus mehreren gleichwertigen Auswahlmöglichkeiten. Nach Minhas ist das Kriterium zur Unterscheidung dieser beiden Arten, ob es unerlässlich ist, alle Antwortmöglichkeiten auf einen Blick zu sehen (Typ I), oder ob es eine sinnvolle Standardantwort gibt, die dem Nutzer vorgegeben werden kann (Typ II). Der erste Typ zeichnet sich durch möglichst komplementäre Optionen aus und lässt sich nach Minhas ideal durch die sogenannten *Radio Buttons* realisieren. Für den zweiten Typ, der aus einer beliebigen Menge an Optionen bestehen kann, würde sich ein *Drop-Down-Menu* eignen. (vgl. MINHAS 2018)

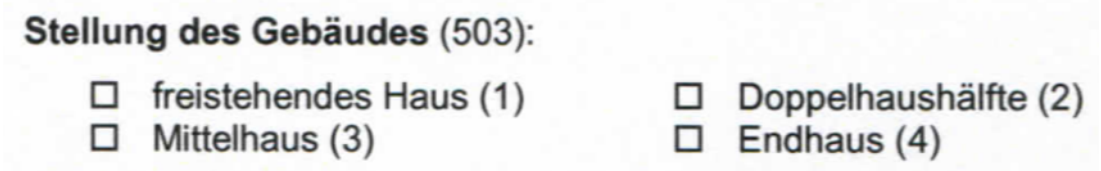


Abbildung 2: Auswahlfelder Typ II (Anlage 1)

Simple Eingabefelder werden schon in dem Fragebogen der GAGs überwiegend vermieden. Stattdessen werden dem Nutzer zahlreiche sinnvolle Antwortmöglichkeiten vorgeschlagen. In Online-Formularen wäre es interessant die Eingaben zu aggregieren und dem Nutzer diese gemäß der Häufigkeit der einzelnen Optionen darzustellen. Das Beispiel in Abbildung 3 könnte aus einer Verbindung zwischen einem normalen Eingabefeld und einem Auswahlfeld umgesetzt werden.

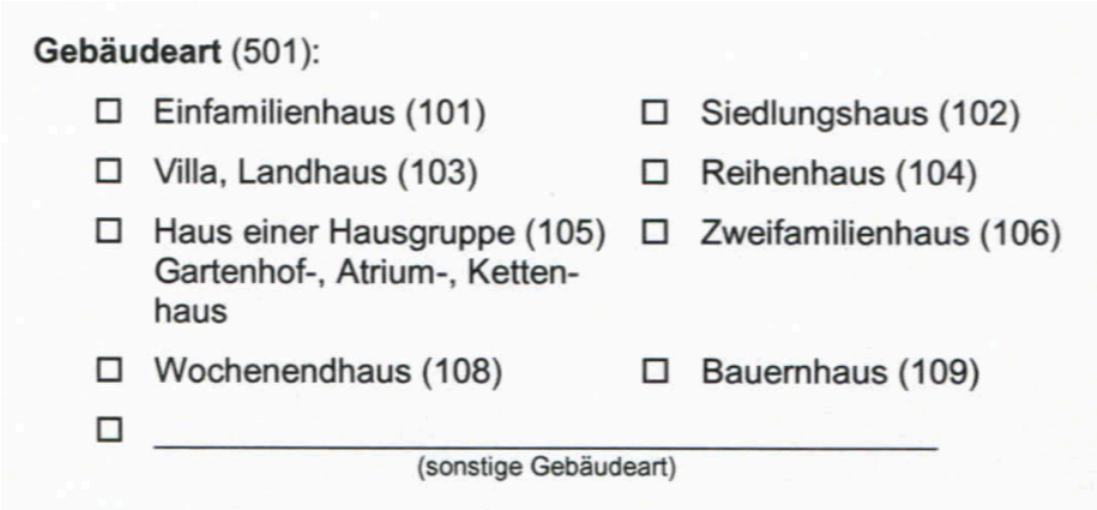


Abbildung 3: Eingabefelder mit Vorschlägen (Anlage 1)

Neben der Eingabe von Texten gibt es auch Felder in denen Zahlenwerte eingetragen werden müssen. Die Abbildung 4 zeigt ein Beispiel für die freie Angabe von quantitativen Werten (Anzahl der Wohnungen im Gebäude). Weitere Beispiele hierfür sind das Baujahr, die Zahl der Vollgeschosse oder prozentuale Angaben.

Zwischen den numerischen Werten und den typischen Auswahlfeldern stehen Daten mit einer natürlichen Reihenfolge („nicht vorhanden“, „einmal vorhanden“, „mehrfach vorhanden“) oder qualitative Daten wie Postleitzahlen.

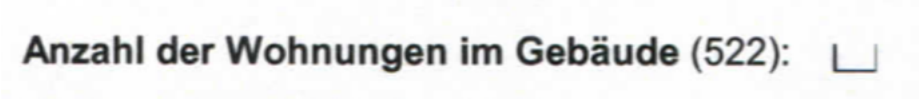


Abbildung 4: Numerische Werte (Anlage 1)

### 2.1.2 Formularlogik

Die Formularlogik in dem Fragebogen der GAGs betrifft alleine die Möglichkeit, aufgrund von Entscheidungsfeldern ganze Abschnitte zu überspringen. Beispielsweise müssen Eigentümer, die ein Abbruchobjekt gekauft haben (Abb. 1), keine Angaben mehr zum Baujahr oder zu den Veränderungen vor dem Erwerb machen (Anlage 1). Komplexere Lösungen sind in analogen Fragebögen kaum zu realisieren.

## 2.2 Anwendungsfälle

Dieser Abschnitt beschreibt die Anwendungsfälle des Fragebogens für Ein- und Zweifamilienhäuser aus der Perspektive der Nutzer und Entwickler um daraus Schlüsse für ein Online-Formular zu ziehen. Aus den Anwendungsfällen leiten sich dann die Anforderungen in Kapitel 2.3 ab.

### 2.2.1 Nutzersicht

Die Menge der vermeintlich hochsensiblen Immobilien- und Käuferdaten in dem Fragebogen der GAGs scheint einigen Eigentümern Sorge zu bereiten. Die Reduzierung der Felder auf ein Minimum kann den Nutzer in dieser Hinsicht beruhigen. Die meisten Daten sind schließlich über die Notare bereits in der Kaufpreissammlung gelandet und müssen nur kontrolliert werden. Aufgrund logischer Schlussfolgerungen können Felder redundant werden. Durch das dynamische Deaktivieren dieser Felder wird eine Zeitersparnis erzielt, die sich wiederum positiv auf die Haltung der Nutzer auswirkt.

Das Online-Formular soll den Nutzer bei der Bearbeitung unterstützen. Freie Eingabefelder können durch statistisch ermittelte Vorschläge erweitert werden, die serverseitig aus beliebig komplexen Kausalzusammenhängen stammen.

Der Trend Mobile First spielt auch bei Online-Formularen eine Rolle. Schlichte Eingabefelder sind gerade auf dem Smartphone eher sperrig zu bedienen und bieten zudem Fehlerquellen.

### 2.2.2 Entwicklersicht

Das Online-Formular ist aus der Entwicklersicht eine Standardaufgabe und sollte daher möglichst einfach in jede Webapplikation implementierbar sein. Einfache Funktionen sollten nicht für jedes Formular neu entwickelt werden. Eine Art Framework kann Redundanzen vermeiden und bestimmte Entwicklungsprozesse automatisieren.

Jede Webanwendung profitiert von der klaren Trennung der Programmlogik von der Darstellung in den Punkten Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit. Beim Hinzufügen von Formularfeldern oder dem Anpassen der Validierung eines Formulars sollten keine Änderungen in der Darstellung vorgenommen werden müssen.

Eine langfristig wartbare Software ist nach Wolff dann sichergestellt, wenn die einzelnen Komponenten eines Systems frei ausgetauscht bzw. neu geschrieben werden können. Systeme, die eine gut dokumentierte Schnittstelle besitzen, können ihre Services und Frameworks jederzeit ersetzen. (vgl. WOLFF 2018: 6)

Der minimale Funktionsumfang eines Online-Formulars muss die in Kapitel 2.1.1 beschriebenen Feldtypen aufweisen. Dazu gehören die unterschiedlichen Auswahlfelder, die Eingabefelder mit Vorschlägen und die Möglichkeit numerische Daten einzugeben. Jedes dieser Feldtypen muss Frontend-seitig modellierbar sein.

## 2.3 Anforderungen

In diesem Kapitel werden die Anforderungen an ein Online-Formular aus den Anwendungsfällen der Nutzer und Entwickler abgeleitet. Dabei wird zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden. Funktionale Anforderungen spezifizieren nach Balzert die Funktionalitäten, die eine Software erfüllen sollte, wohingegen nicht-funktionale Anforderungen die technischen Anforderungen betreffen (vgl. BALZERT 2011: 109).

### 2.3.1 Funktional

Das Online-Formular soll auf einfache und komplexe Formulare angewendet werden können. Es werden im Mindestumfang unterschiedliche Auswahlfelder, Eingabefelder mit Vorschlägen und numerische Werte implementiert. Alle Felder können beliebig validiert werden und reagieren mit angemessenen Fehlermeldungen. Auf dem Server können zudem die Felder festgelegt werden, über die eine serverseitige Validierung initiiert werden soll.

Felder können jederzeit deaktiviert oder verborgen werden, wenn sie aufgrund der Programmlogik redundant sind. Dem Nutzer wird damit suggeriert, dass auf die effiziente Bearbeitung des Formulars Wert gelegt wird.

Die Programmlogik im Backend und die Darstellung des Formulars im Frontend werden gegen eine festgelegte und versionierte Web-API modelliert. Daher ist es problemlos möglich serverseitig neue Felder zu integrieren oder die Validierung im laufenden Betrieb anzupassen.

Eingabefelder werden durch Vorschläge unterstützt, die aus beliebigen Datenquellen zusammengesetzt werden können. Der Entwickler hat freie Hand bei der Gestaltung der Programmlogik.

### 2.3.2 Nicht-Funktional

Das Online-Formular soll die Komplexität eines analogen Formulars bei gleichbleibendem Informationsgehalt reduzieren. Jede Implementierung im Frontend muss dafür die gesamte Funktionalität abbilden, sodass das Design frei austauschbar ist.

Weitere typische nicht-funktionale Anforderungen wie die Sicherheit des Systems (Authentifizierung und Autorisierung), Performanceaspekte oder die Ausfallsicherheit sollen nicht betrachtet werden, da diese nicht Teil der Arbeit sind.

# 3 Konzept

In der Regel besteht jede Webapplikation aus einem Frontend und einem Backend. Das Frontend ist die Benutzeroberfläche und damit der Teil der Anwendung, der über den Webbrowser mit dem Nutzer interagiert. Der Webbrowser kann HTML- und CSS-Dateien darstellen und über einen Interpreter auch JavaScript-Code ausführen. Das Backend ist für die Datenhaltung und sämtliche Berechnungen zuständig, die direkt beim Client nicht performant genug wären. Das Frontend stellt Anfragen an den Server (Backend) und visualisiert die Antworten in der Benutzeroberfläche. (vgl. BITKOM 2015: 5)

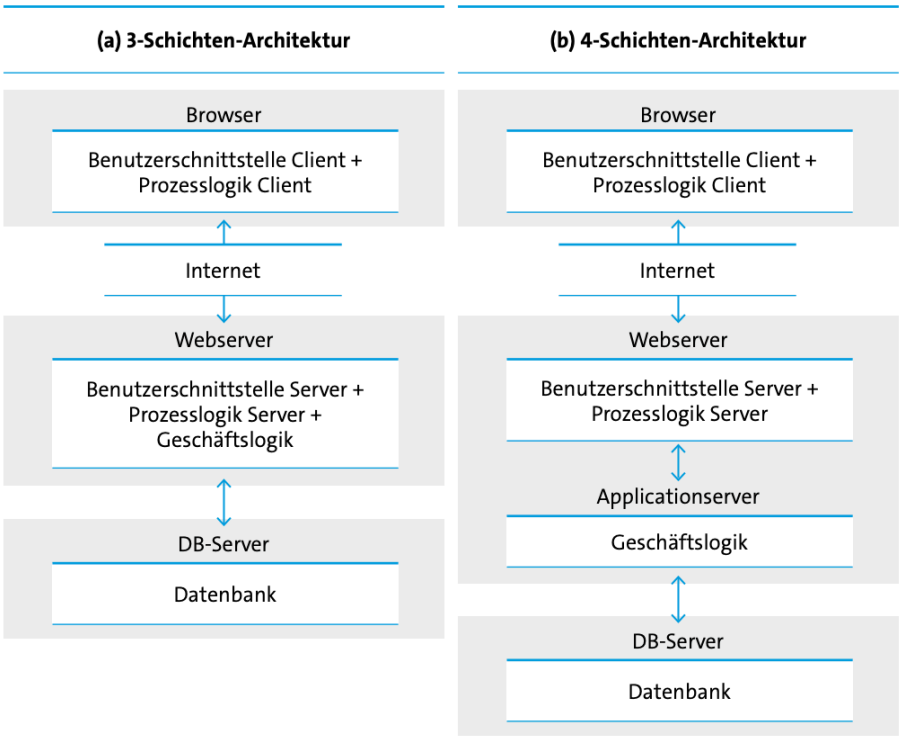


Abbildung 5: 3- und 4-Schichten-Architektur (BITKOM 2015: 6)

In der Abbildung 5 werden das 3- und 4-Schichten-Architekturmodell miteinander verglichen. Der Unterschied ist die klare Trennung des Webservers von der Geschäftslogik. In dieser Arbeit wird das 3-Schichten-Modell genutzt, da keine Anwendung für die Produktionsumgebung entwickelt werden muss. Zudem kann von einem serverzentrierten Ansatz gesprochen werden, da die gesamte Formularlogik auf dem Server realisiert wird. (vgl. BITKOM 2015: 7)

## 3.1 Online-Formular

Wie bereits beschrieben, ist ein Ziel dieser Arbeit die Möglichkeiten der Technologien Angular, gRPC und Go im Rahmen der Entwicklung eines Online-Formulars zu erörtern. In diesem Kapitel wird auf die Funktionen eingegangen, die diese in den Bereichen Frontend und Backend einnehmen.

### 3.1.1 Frontend

Das Frontend ist die Benutzeroberfläche, über die Clients mit einem Server kommunizieren. Die wichtigsten Technologien in diesem Bereich sind HTML, CSS und JavaScript. Angular ist ein JavaScript-Framework, das hauptsächlich auf der Programmiersprache TypeScript aufbaut. TypeScript kompiliert in JavaScript und bietet zudem die Möglichkeit weitere Stylesheet-Sprachen wie Sass oder Less (dt. weniger) über CSS-Präprozessoren zu nutzen.

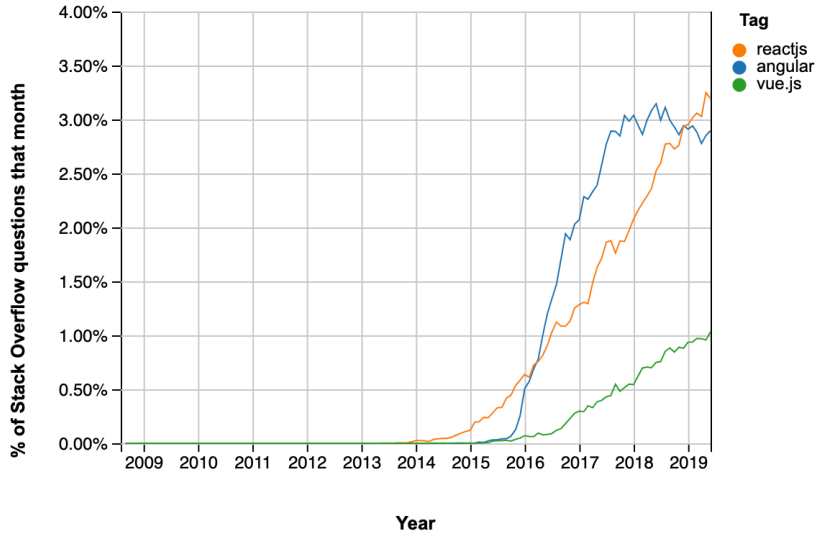


Abbildung 6: JavaScript-Frameworks (STACKOVERFLOW 2019)

In der Abbildung 6 werden die drei größten JavaScript-Frameworks anhand der Anzahl der Fragen auf StackOverflow zwischen 2000 und 2019 miteinander verglichen. Angular scheint sich dabei mit React.js den Platz des populärsten JavaScript-Frameworks zu teilen. Es ist weit verbreitet und wird als Open Source Projekt von Google und einer Online-Community weiterentwickelt. (vgl. ALEXSOFT 2019)

Mit Angular ist es möglich kleine und große Webapplikationen zu realisieren und langfristig zu warten. Dies wird durch die Unterstützung der Programmiersprache TypeScript und die komponentenbasierte Architektur erreicht. Zudem bietet Angular eine ausgefeilte Template-Syntax, bestehend aus Direktiven für die Veränderung des DOM (Document Object Model) sowie das *Two-Way Data Binding* für die beidseitige Verknüpfung des Templates mit einer Variable. (vgl. ALEXSOFT 2019)

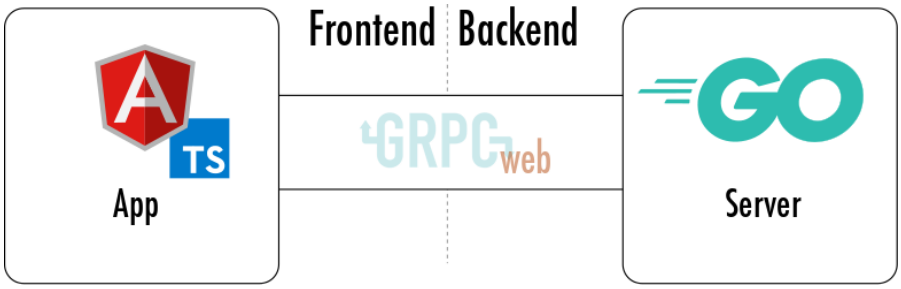


Abbildung 7: Client-Server-Modell über gRPC (eigene Darstellung)

Das Online-Formular benötigt keine öffentliche Web-API. Daher eignen sich auch andere Technologien als REST für die Kommunikation zwischen dem Frontend und dem Backend. In dieser Arbeit wird das gRPC-Verfahren genutzt (Abb. 7). Es basiert auf der Idee einen Service mit Methoden, Parametern und Rückgabewerten zu definieren (Procedure) und diesen entfernt aufrufen zu können (Remote Call). (vgl. INDRASIRI & SIRIWARDENA 2018: 68)

### 3.1.2 Backend

Im Backend wird auf die Programmiersprache Go gesetzt. Der Go Server, der in den Abbildungen 7 und 8 dargestellt ist, nimmt die Anfragen vom Client (App) entgegen und ist für die Geschäftslogik verantwortlich. Diese beinhaltet die Validierung und die Modellierung des Formulars.

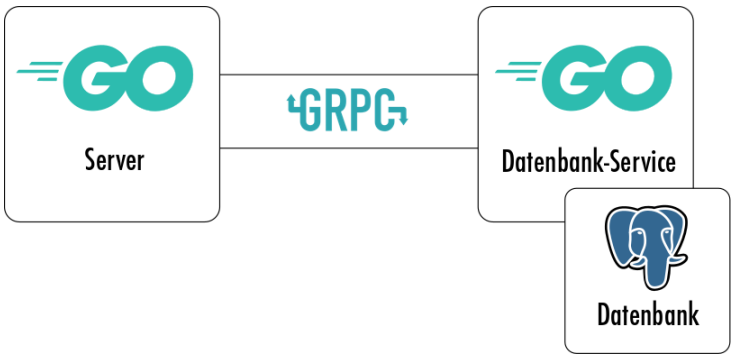


Abbildung 8: Services-Kommunikation über gRPC (eigene Darstellung)

In einer Microservices-Architektur werden Datenbanken über einen Datenbank-Service angesprochen (Abb. 8). Dadurch wird gewährleistet, dass diese nicht von anderen Services korrumpiert werden können. Die Kommunikation zwischen den Backend-Services ist wieder ein typisches Anwendungsgebiet der gRPC-Technologie.

## 3.2 Entwicklungsumgebung

Mit Docker hat sich das Aufsetzen einer Entwicklungsumgebung vollständig gewandelt. Jede Programmiersprache und viele Anwendungen bieten Docker Images an, die für die Erzeugung von Docker Containern genutzt werden, in denen die Programme betriebssystemunabhängig laufen. Docker Compose ist eine Software von Docker, die es erlaubt, viele Docker Container in einem gemeinsamen Netzwerk zu starten.

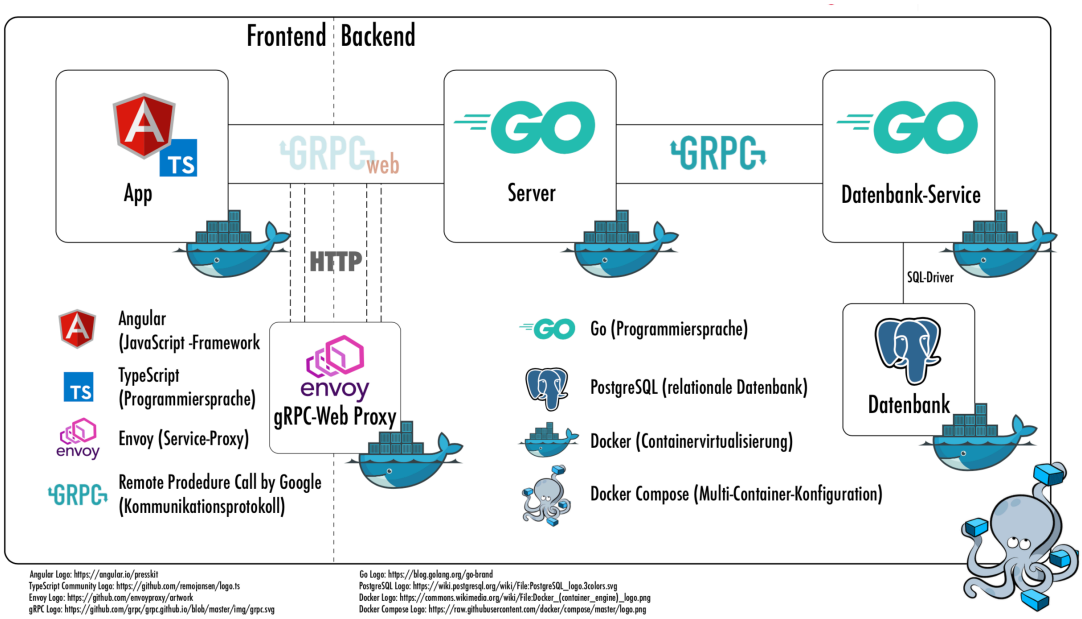


Abbildung 9: Entwicklungsumgebung mit Docker (eigene Darstellung)

Die Abbildung 9 zeigt die Entwicklungsumgebung für das Online-Formular. Die einzelnen Komponenten werden in Docker Container verpackt und kommunizieren über das Netzwerk, welches von Docker Compose bereitgestellt wird.

# 4 Grundlagen

In diesem Kapitel werden die Grundlagen für das Verständnis der Technologien gelegt, die für die Umsetzung des Online-Formulars in Kapitel 5 von Nutzen sind. Dabei wird das Augenmerk auf die Funktionen gelegt, die für das Online-Formular relevant sind. Es werden ausdrücklich keine Einführungen in die Technologien gegeben, sondern nur die für dieses Projekt wesentlichen Eigenschaften erläutert.

Neben Angular, Docker, gRPC und Go kommen am Rande noch weitere Technologien zur Sprache. Dabei ist der gRPC-Client für TypeScript und die Software Envoy zu nennen. Zu den Erläuterungen werden jeweils zusammenhängende Code-Beispiele und Kommandozeilenbefehle geliefert.

## 4.1 gRPC

gRPC ist ein hoch performantes RPC-Framework, dass in jeder Software-Umgebung laufen kann. Google hat das hausinterne RPC-System mit dem Namen Stubby (dt. stummelig) im Jahr 2015 als ein Open Source Projekt veröffentlicht (vgl. MENGE-SONNENTAG 2016). Im August 2016 wurde dann die Version 1.0.0 mit den Programmiersprachen C, Python, Ruby, PHP, C# und Objective-C herausgebracht (vgl. NOBLE 2016).

Selbstverständlich gibt es auch eine gRPC-Implementierung für Googles Programmiersprache Go, die das Erstellen eines gRPC-Servers bzw. gRPC-Clients (auch *Stub* genannt) ermöglicht.

Am 28. Oktober 2018 wurde bekanntgegeben, dass gRPC nun auch für das Web verfügbar ist (vgl. PERKINS, CHEUNG & SETHURAMAN 2018a). Damit können Programmierschnittstellen in den Sprachen JavaScript und TypeScript erstellt werden. Besonders für TypeScript ist die Entwicklung interessant, da es die Möglichkeit bietet, zwischen dem Server und dem Client typsicher zu kommunizieren.

### 4.1.1 Protocol Buffers

Das gRPC-Framework nutzt standardmäßig Protocol Buffers (Protobuf). Dies ist Googles Protokollsprache für die flexible, automatisierte und effiziente Verarbeitung strukturierter Daten (vgl. INDRASIRI & SIRIWARDENA 2018: 68). Der Protocol Buffers Compiler, auch Proto Compiler (*protoc*) genannt, liest das Format ein und generiert Programmcode für Server und Client in vielen verschiedenen Programmiersprachen. Eine Besonderheit des Proto Compilers ist es, dass Änderungen an der Datenstruktur möglich sind, ohne die Schnittstelle zu korrumpieren. Datentypen werden mit einem numerischen Wert belegt und so von dem Compiler automatisch in die Codeerzeugung mit einbezogen (vgl. GOOGLE 2019).

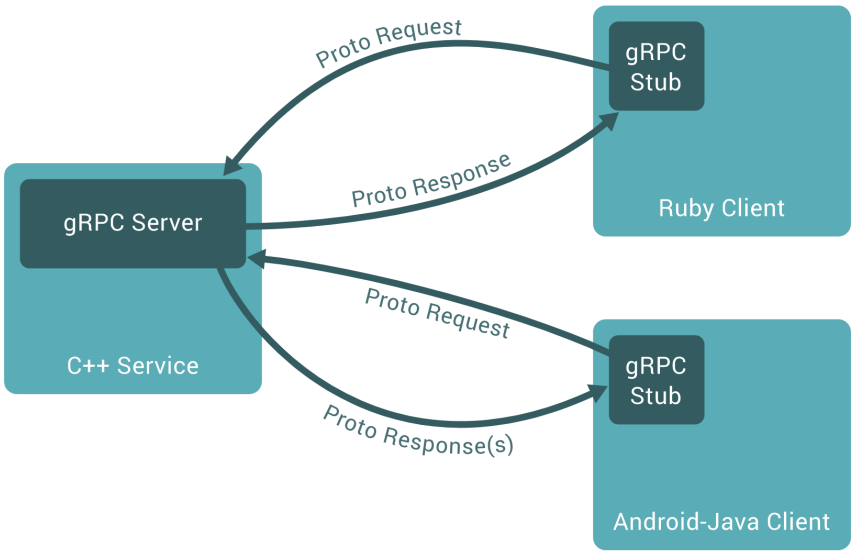


Abbildung 10: gRPC-Prinzip (GRPC 2019)

Die Idee hinter dem Verfahren ist die Spezifizierung eines Services mit seinen Methoden, Parametern und Rückgabewerten sowie die Bereitstellung sogenannter *Stubs* mit dem der Client diese Methoden über einen Remote Call aufrufen kann. Die Abbildung 10 zeigt einen gRPC-Server in der Programmiersprache C++. Die Client-*Stubs* sind in den Sprachen Ruby und Android-Java implementiert. Diese senden einen Proto Request an den Server und erhalten die Proto-Response als Rückgabewert. (vgl. GRPC 2019)

```
service Example {
    rpc GetExample (GetExampleRequest) returns (Example) {}
}

message GetExampleRequest {
    int64 id = 1;
}

message Example {
    int64 id = 1;
    ...
}
```

Code 1: Protocol Buffer Beispiel

Code 1 zeigt die Definition eines *Example*-Services mit der Methode *GetExample*. Die Parameter und die Rückgabewerte werden jeweils in einer Message ausgedrückt. Je nachdem welche Programmiersprache genutzt wird, wird z.B. ein *Example*-Struct in Go oder eine *Example*-Class in TypeScript oder Java zurückgegeben.

In den folgenden Kapiteln wird aus der Definition in Code 1 ein TypeScript gRPC-Web-Client und ein gRPC-Server in Go erstellt.

| .proto Type | Notes | C++ Type | Java Type | Python Type[2] | Go Type | Ruby Type | C# Type | PHP Type |
|---|---|---|---|---|---|---|---|---|
| double | | double | double | float | float64 | Float | double | float |
| float | | float | float | float | float32 | Float | float | float |
| int32 | Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead. | int32 | int | int | int32 | Fixnum or Bignum (as required) | int | integer |
| int64 | Uses variable-length | int64 | long | int/long[3] | int64 | Bignum | long | integer/string[ |

Abbildung 11: Korrespondierende .proto Datentypen (GRPC 2019)

Der Ausschnitt aus der Tabelle in der Dokumentation des Protocol Buffers zeigt die korrespondierenden Datentypen der unterschiedlichen Programmiersprachen zum Proto-Type (Abb. 11). Die Proto-Types können also in allen Sprachen sinnvoll repräsentiert werden.

### 4.1.2 gRPC-Web und Envoy

Eine gRPC-Service-Architektur bezieht auch die Kommunikation über das Internet mit ein. Das gRPC-Konzept ermöglicht das Ausführen von RPC-Aufrufen über die Web-Schnittstelle. Allerdings ist dies, wie in der Abbildung 12 zu sehen, nur über eine Brückentechnologie möglich. Envoy ist ein Open Source Proxy, der HTTP/1.1-Aufrufe von einem Client in HTTP/2-Aufrufe übersetzten kann. HTTP/2 ist das Standardprotokoll von gRPC für den Transport ihrer Remote Calls (vgl. PERKINS 2018b).

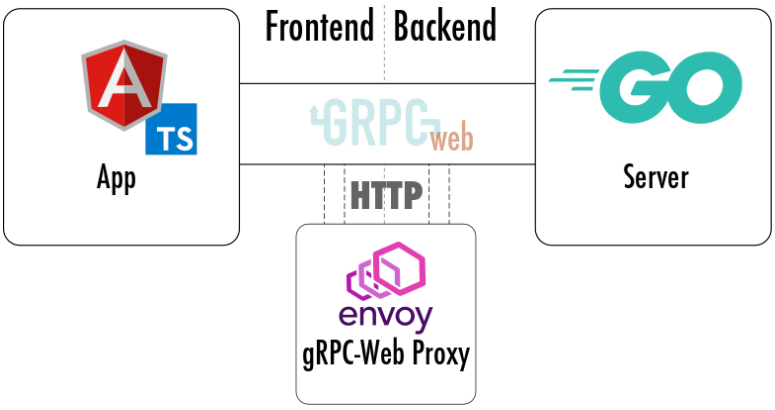


Abbildung 12: gRPC-Web Realisierung mit Envoy (eigene Darstellung)

Die meisten Webanwendungen nutzen das REST-Paradigma für die Interaktion zwischen Server und Client (Abbildung 13). Die *GET*- und *POST*-Methoden eines REST-Aufrufes werden von einer RESTful API übersetzt, damit die Services im Hintergrund über das gRPC-Verfahren kommunizieren können. Im Gegensatz dazu legt gRPC-Web einen Layer über die weiterhin bestehenden HTTP-Aufrufe und ermöglicht so eine scheinbar HTTP-freie gRPC-Service-Architektur. (vgl. PERKINS 2018b)

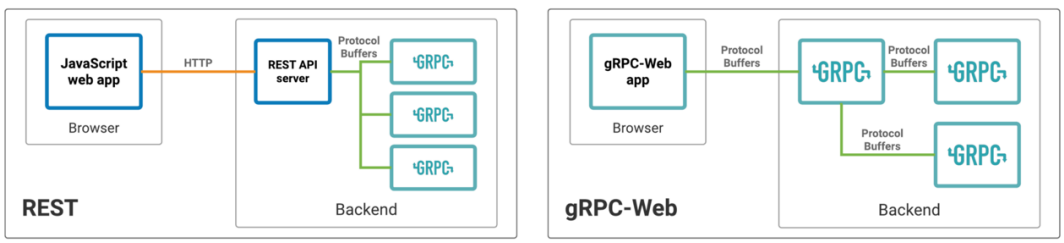


Abbildung 13: REST vs. gRPC (PERKINS 2018b)

## 4.2 Angular

Das Open Source Projekt Angular wurde im September 2016 in der Version 2.0.0 von Google veröffentlicht und hat sich seitdem neben React.js und Vue.js zu einem der populärsten JavaScript-Frameworks der Welt entwickelt. Es nutzt die statisch-typisierbare Programmiersprache TypeScript und hat ein ausgereiftes Kommandozeilenprogramm mit dem sämtliche Komponenten erstellt, Anwendungen kompiliert und über einen lokalen Server bereitgestellt werden können. (vgl. MALCHER, HOPPE & KOPPENHAGEN 2019: vii)

Für JavaScript-Frameworks wie Angular, React.js und Vue.js ist das responsive Design absoluter Standard. Die Webseiten passen sich problemlos an jeden Viewport an und haben Lösungen für sämtliche Browser parat. Das Besondere dieser Frameworks ist aber die Single Page Application. Es werden keine HTML-Seiten mehr entwickelt, sondern Module und Komponenten, die Daten bereithalten, welche dem Client je nach Nutzerinteraktion bereitgestellt werden können. Eine Standardfunktionalität von JavaScript-Frameworks ist zudem die Implementierung von *Service-Workern*, die es ermöglichen, große Datenmengen direkt beim Client zu speichern, um die Webapplikation auch ohne Internetzugang weiter nutzen zu können (Progressive Web App).

Angular sollte nicht mit AngularJS verwechselt werden. Seit der Version 2.0.0 hat das Angular-Framework nicht nur den Suffix JS abgelegt, sondern wurde von Google auch komplett überarbeitet. Angular ist daher nicht mehr mit dem alten AngularJS kompatibel. (vgl. MALCHER et al. 2019: ix)

In den folgenden Kapiteln werden neben der Programmiersprache TypeScript die Angular Components, die Template Syntax, die Angular CLI und die Entwicklung von Angular Libraries vorgestellt. Dabei wird davon ausgegangen, dass Node.js bzw. der Node Package Manager (NPM) bereits installiert ist. (NODE.JS.ORG 2019)

### 4.2.1 Node.js & NPM

Node.js ist eine Laufzeitumgebung, die das Ausführen von JavaScript auf einem Server ermöglicht. Es basiert auf der Google V8 Engine, die auch in Google Chrome Verwendung findet (NODE.JS.ORG 2019). In diesem Projekt werden eine Vielzahl an Node.js-Paketen genutzt sowie eigene Pakete mit dem Node Package Manager in die NPM-Online-Registry hochgeladen. NPM kümmert sich dabei um alle Abhängigkeiten zu Bibliotheken, die in einem Projekt vorkommen und speichert diese versioniert ab.

```
npm install ts-protoc-gen
```

Befehl 1: Installation des TypeScript-Plugins (IMPROBABLE-ENG 2019)

Mit dem Befehl 1 kann über NPM beispielsweise das TypeScript-Plugin für den Protocol Buffers Compiler installiert werden. Dabei werden die Abhängigkeiten dieser Bibliothek automatisch zum Projekt hinzugefügt.

### 4.2.2 gRPC-Client für TypeScript

Das Open-Source-Projekt gRPC-Web von Improbable-Engineering bietet ein TypeScript-Plugin für den Protocol Buffers Compiler sowie eine TypeScript-Bibliothek für die Erstellung eines gRPC-Clients an. Diese beiden Programme werden für die Umsetzung der Web-API genutzt. Nachdem der Proto Buffers Compiler heruntergeladen wurde (vgl. GOOGLE 2019) kann mit dem Befehl 1 das TypeScript-Plugin installiert werden. Die ausführbare Datei des Plugins befindet sich in dem *Node-Modules*-Ordner des Projekts. (vgl. IMPROBABLE-ENG 2019)

```
protoc \
--plugin="protoc-gen-ts=${PROTOC_GEN_TS_PATH}" \
--js_out="import_style=commonjs,binary:path/to" \
--ts_out="service=true:path/to“ \
example.proto
```

Befehl 2: Generieren eines TypeScript-Stubs

Mit dem Befehl 2 wird dann ein TypeScript-*Stub* (gRPC-Client) erzeugt. Der generierte Code kann nun von jedem TypeScript-Projekt importiert und für Requests an einen gRPC-Server genutzt werden.

```
import {
  ExampleServiceClient,
  GetExampleRequest
} from "path/to/example_pb_service";

const client = new ExampleServiceClient(“gRPC-Server-Adresse“);
const req = new GetExampleRequest();
req.setId(100);
client.getExample(req, (err, example: Example) => {
        //
});
```

Code 2: Beispiel-Request vom Client

Code 2 zeigt ein Beispiel für die Nutzung des gRPC-Clients mit TypeScript. Es wird die Methode *getExample* aufgerufen, die im Protobuf aus Code 1 definiert wurde. Zurückgegeben wird die TypeScript-Klasse *Example*. Die Typsicherheit von TypeScript kann so in einer Angular-App genutzt werden.

### 4.2.3 Angular CLI

Die Angular CLI (ng) ist das Kommandozeilenprogramm von Angular, das den gesamten Entwicklungsprozess von dem Erstellen der Komponenten über das Testen bis zur Kompilierung eines Angular-Projektes übernimmt.

```
import {
  ExampleServiceClient,
  GetExampleRequest
} from "path/to/example_pb_service";

const client = new ExampleServiceClient(“gRPC-Server-Adresse“);
const req = new GetExampleRequest();
req.setId(100);
client.getExample(req, (err, example: Example) => {
        //
});
```

Code 3: Benutzen der gRPC-Schnittstelle mit TypeScript

Das Programm kümmert sich um einen Großteil der wiederkehrenden Arbeiten in einem Projekt und sorgt für eine konsistente Struktur (vgl. MALCHER et al. 2019, 21-22). Der folgende Abschnitt widmet sich den wichtigsten Befehlen dieses Programms.

```
ng new PROJECT-NAME --routing true --style css
cd PROJECT-NAME
ng serve --open
```

Befehl 3: Erstellen und Starten einer Angular-App

Mit dem Befehl 3 wird ein neues Angular-Projekt in einem Unterordner angelegt. Der Parameter Routing sorgt dafür, dass die Single Page Application URI- (Uniform Resource Identifier) Pfade auslesen und dazu unterschiedliche Komponenten aufrufen kann. Zudem werden alle Stylesheets als CSS-Dateien angelegt. Mit *Serve* und *Open* wird die Angular-App gestartet und im Browser unter der Adresse *localhost:4200* geöffnet.

```
ng generate component my-component
ng g service my-service
ng g m my-module
```

Befehl 4: Erstellen von Components, Services und Modules

Die Angular CLI nimmt viele Arbeiten wie das Erstellen von Komponenten, Services und Modules ab (Befehl 4) und sorgt für eine saubere Integration in die Angular-App.

### 4.2.4 Angular Components & Template-Syntax

Die Angular Components sind der Grundbaustein jeder Angular-App. Sie bestehen aus einer TypeScript-Klasse, einem HTML-Template und einem Stylesheet. Angular Components verbinden die Logik einer Applikation mit der Benutzeroberfläche und interagieren darüber mit dem Nutzer (vgl. MALCHER et al. 2019: 69-71).

Angular verwendet zusätzlich zu den gewohnten HTML-Ausdrücken eine Template-Syntax, die es ermöglicht, Variablen innerhalb der TypeScript-Klasse mit Ausdrücken in dem Template zu verbinden. Dabei gibt es verschiedene Möglichkeiten, die im folgenden Abschnitt kurz erläutert werden.

```
@Component({
  selector: 'app-root',
  template: `
        Welcome to {{title}}!
        <button [disabled]="isDisabled" (click)="do()">Disabled Button</button>
  `,
  styles: []
})
export class AppComponent {
  title: string = 'example';
  isDisabled: boolean = true;
  do(): void {
    // Click Event
  }
}
```

Code 4: Interpolation, Property- und Event-Bindings

Der Template-Ausdruck innerhalb der geschwungenen Klammern in Code 4 wird ausgewertet und als *string* (Text) ausgegeben. Das wird Interpolation genannt. Innerhalb der Interpolation können nahezu beliebige Berechnungen durchgeführt werden. Im Gegensatz dazu müssen andere Datentypen wie *boolean* (Datentyp für Wahrheitswerte) mithilfe sogenannter Property-Bindings angegeben werden. Bei diesen werden die Attribute der DOM-Elemente mit eckigen Klammern umrandet (*disabled*). Ausgegeben wird in diesem Fall der Text „Welcome to example“ und ein deaktivierter Button.

Events, wie die Bestätigung eines Buttons oder die Eingabe in ein Textfeld können mit Event-Bindings abgefangen werden. Das Klick-Event in Code 4 wird automatisch mit der Funktion *do* verknüpft (vgl. MALCHER et al. 2019: 77-78).

Die Besonderheit von Interpolation und Property-Bindings ist die automatische Aktualisierung des Templates, sobald sich die Variable in der TypeScript-Klasse ändert. Mit der Template-Syntax „[(…)]“ lassen sich Variablen und DOM-Attribute sogar beidseitig verbinden (*Two-Way Data Binding*). Damit werden bei Nutzereingaben, die über HTML in die DOM-Elemente geschrieben werden, automatisch in die Variablen der TypeScript-Klasse aktualisiert (vgl. MALCHER et al. 2019: 79).

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

@Component({
  selector: 'app-root',
  template: `
        <ul>
          <li *ngFor="let name of names; index as i">
            <p *ngIf="i < 3">
              {{ name }}
            </p>
          </li>
        </ul>
  `,
  styles: []
})
export class AppComponent {
  names = ["Alex", "Bela", "Chris", "Dirk", "Eric"]
}
```

Code 5: Strukturdirektiven ngFor und ngIf

Code 5 zeigt, wie mithilfe von *ngFor*-Direktiven über ein *Array* iteriert wird. Dabei werden die Indices der *Array*-Elemente mit der Variable *i* verknüpft. Die *ngIf*-Direktive stellt eine Bedingung auf, zu der das *p*-Tag mit der Interpolation in das DOM eingefügt wird. Dieses Beispiel zeigt eine Liste mit den ersten drei Namen des *Arrays*.

### 4.2.5 Angular Library

Viele Applikationen haben ähnliche Ziele und benötigen daher auch ähnliche Komponenten und Services. Angular bietet hierfür Bibliotheken an, die mit NPM publiziert und installiert werden können. Mit dem Befehl 5 wird innerhalb eines Angular-Projektes eine Angular Library erstellt. Die Angular CLI sorgt dafür, dass diese von einem lokalen Projekt während der Entwicklung so genutzt werden kann wie externe Bibliotheken. (vgl. ANGULAR 2019)

```
ng generate library my-library
ng build my-library
```

Befehl 5: Erzeugen einer Angular Library

Das Verpacken von Logikbausteinen einer App in Libraries ermöglicht das Wiederverwenden dieser in anderen Anwendungen. So können mit Angular sehr komplexe Anwendungen gebaut werden, die trotzdem wartbar, erweiterbar und lesbar bleiben. Die NPM-Online-Registry eignet sich für die Veröffentlichung der Libraries, um diese im Unternehmen und mit anderen Entwicklern auf der ganzen Welt zu teilen. (vgl. ANGULAR.IO 2019)

```
cd dist/my-library
npm publish
```

Befehl 6: Veröffentlichen einer Angular Library

Bevor eine Angular Library mit dem Befehl 6 in der NPM-Online-Registry veröffentlicht wird, sollte die TypeScript-Konfigurationsdatei mit dem Namen, der Version und einer Beschreibung gefüllt werden, um anderen Entwickler einen Überblick über den Funktionsumfang der Bibliothek zu geben. Wichtig ist es alle Abhängigkeiten der Bibliothek zu spezifizieren. So werden diese Abhängigkeiten bei der Installation (Befehl 1) eingeblendet.

## 4.3 Go

Go oder auch Golang, wie es suchmaschinenkompatibel auch genannt wird, ist eine kompilierbare Programmiersprache, die sich an C orientiert (vgl. WOLFF 2018: 69), aber eine klarere Syntax und ein besseres Package-Management-System bietet. Es hat zudem einen Garbage Collector, unterstützt echte Nebenläufigkeit über sogenannte Goroutines und Channels und wurde in Hinblick auf eine möglichst schnelle Kompilierungszeit entwickelt. (vgl. PATEL 2017)

Go wurde im Jahr 2009 als Open Source Projekt von Google veröffentlicht und ging im März 2012 mit der Alpha-Version online. Seitdem wird es hauptsächlich für die Entwicklung von Webapplikationen, CLIs und Microservices verwendet (vgl. GERRAND 2012). Zu den größten Nutzern der Sprache gehören Uber, Twitch, Dailymotion, Medium und selbstverständlich Google (vgl. GOWITEK 2018).

```go
package main

import (
   "io"
   "log"
   "net/http"
)

func main() {
   // Hello world, the web server

   helloHandler := func(w http.ResponseWriter, req *http.Request) {
      io.WriteString(w, "Hello, world!\n")
   }

   http.HandleFunc("/hello", helloHandler)
   log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Code 6: Einfacher Webserver (GOLANG 2019e)

Als Ersatz für das typische Hello-World-Beispiel zeigt Code 6 die Implementierung eines einfachen Webservers in Go. Dabei werden alleine die Standardbibliotheken io, log und net/http benötigt. Im nächsten Kapitel werden die Besonderheiten der Programmiersprache erläutert, dann folgt die Implementierung eines gRPC-Servers.

### 4.3.1 Besonderheiten

Go verfügt über eine ausgeprägte Standardbibliothek (*batteries included*). Besonders im Bereich der Netzwerkprogrammierung (Code 6) reicht die Standardbibliothek meist aus. Auf der Website der Programmiersprache sind alle Bibliotheken mit startbereiten Beispielen dokumentiert. (vgl. GOLANG 2019b)

Im Gegensatz zu den meisten anderen Programmiersprachen besitzt Go keine expliziten Interfaces und hat keine Vererbung. Jeder Typ in Go hat ein Interface, dass über seine Methoden definiert wird. Diese impliziten Interfaces werden genau dann definiert wenn sie gebraucht werden. Um ein Interface in Go zu erfüllen, müssen alleine die Methoden dieses Interfaces implementiert werden. (vgl. TAYLOR 2009)

```
Read(b []byte) (n int, err error)
```

Code 7: io.Reader Interface (GOLANG 2019c)

Code 7 zeigt die Methode des io.Reader Interfaces. Dieses Interface wird in der Standardbibliothek von vielen Typen implementiert (vgl. GOLANG 2019f). Dadurch können Funktionen, die das io.Reader Interface als Parameter akzeptieren, Daten aus den verschiedensten Ressourcen nutzen. Mithilfe der Standardbibliothek *ioutil* kann ein Byte-Slice (eine Art Array mit einer dynamischen Größe) von jedem beliebigen Reader zurückgegeben werden (Code 8).

```
func ReadAll(r io.Reader) ([]byte, error)
```

Code 8: Nutzen des Reader Interfaces (GOLANG 2019d)

Wie in Code 7 bzw. Code 8 zu sehen, gibt es in Go die Möglichkeit beliebig viele Rückgabewerte zu liefern. Der letzte Parameter ist dabei konventionell ein Typ, der das Error-Interface implementiert (Code 9).

```
Error() string
```

Code 9: Error Interface

Go wurde für echte Nebenläufigkeit entwickelt und nutzt hierfür sogenannte Goroutines (Code 10) und Channels, die viel leichtgewichtiger agieren können

als vergleichbare Mechanismen in anderen Sprachen wie z.B. Threads in Java. (vgl. PIKE 2013)

```
go func() {
   // Goroutine
}()
```

Code 10: Beispiel einer Goroutine

Die Lesbarkeit von Code wird in Go durch *gofmt* unterstützt. Das Tool bewirkt, dass jeder Code auf die gleiche Weise formatiert wird (vgl. GERRAND 2012). Damit werden Diskussionen z.B. um die Einrückung mit Leerzeichen oder Tabulatoren im Keim erstickt und die Entwicklung anderer Go Tools vereinfacht.

Go bietet die Möglichkeit statisch gelinkte, ausführbare Dateien für viele Betriebssysteme zu erstellen. Dafür reicht es, das Betriebssystem und die Mikroprozessor-Architektur zu spezifizieren und diese wie in Befehl 7 an den go build Befehl zu übergeben (vgl. GOLANG 2019g). Die statisch erzeugten Binaries benötigen keine weiteren Abhängigkeiten und können daher in den kleinstmöglichen Linux-Containern von Docker laufen (vgl. WOLFF 2018: 90).

```
GOOS=linux GOARCH=amd64 go build path/to/package
```

Befehl 7: Erzeugen einer Linux amd64 Binary

Ein Pointer (dt. Zeiger) ist ein Wert, der auf die Speicheradresse von einem anderen Wert zeigt. Diesen Mechanismus hat Go von der Programmiersprache C übernommen und grenzt es von anderen typisierten Sprachen wie Java ab, in der alle Werte Referenzen auf Speicheradressen sind. In Go ist es daher möglich, die Kopie von einem Objekt als Wert oder die Speicheradresse eines Wertes über einem Pointer zu übergeben. (vgl. CHENEY 2017)

### 4.3.2 gRPC-Server für Go

```
protoc \
        -I:. \
        --go_out=plugins=grpc:. \
        ./example.proto
```

Befehl 8: Generieren eines gRPC-Servers in Go

Das gRPC-Plugin des Protocol Buffers Compiler sorgt bei der Generierung des Go Codes mit dem Befehl 8 für die Erstellung eines Interfaces, dass von einem Go Type für die Nutzung als gRPC-Server implementiert werden muss. Dem Beispiel Projekt aus Code 1 folgend, muss der Backend-Server nun die Methode *GetExample* vom *ExampleServer* Interface (Code 11) aufweisen, um Anfragen von einem *ExampleClient* erhalten zu können.

```
type ExampleServer interface {
   GetExample(context.Context, *GetExampleRequest) (*Example, error)
}
```

Code 11: Interface des Example-Servers in Go

## 4.4 Docker

Im Gegensatz zu einem Monolithen besteht eine Microservices-Architektur aus vielen einzelnen Services, die eine eigene Entwicklungsumgebung benötigen. Ohne das Verpacken von Anwendungen in Linux-Containern und die damit verbundene Virtualisierung von Anwendungen wäre die Entwicklung solcher komplexen Strukturen viel aufwändiger. Docker hat mit dem Dockerfile für das Aufsetzen dieser Container einen Standard entwickelt, der sich in der Branche durchgesetzt hat.

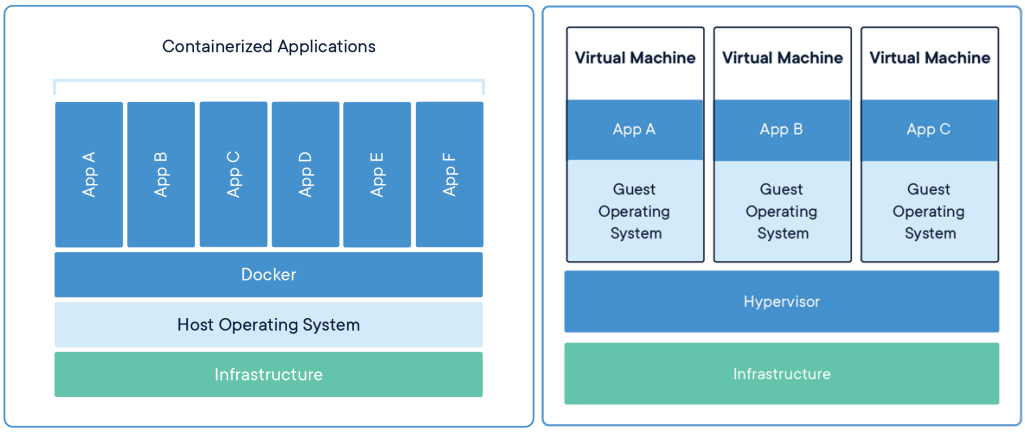


Abbildung 14: Docker vs. virtuelle Maschinen (Docker 2019)

Anders als virtuelle Maschinen nutzt Docker den Kernel des Betriebssystems auf dem Host. Dadurch sind Docker Container nicht so aufwändig zu betreiben und viel leichtgewichtiger als virtuelle Maschinen (Abb. 14). In jedem Docker Container läuft nur ein einzelner Prozess, sodass es schon mit einem einfachen Laptop möglich ist, hunderte Container gleichzeitig zu starten (vgl. WOLFF 2018: 63).

Docker ist eine Linux-Technologie, die aber auch auf anderen Betriebssystemen genutzt werden kann. Auf MacOS und Windows laufen die Linux-Programme auf virtuellen Maschinen mit einer Linux-Umgebung (vgl. WOLFF 2018: 65).

### 4.4.1 Dockerfiles & Docker Images

Eine Dockerfile definiert alle Abhängigkeiten und Befehle, die nötig sind, um ein Docker Image zu erstellen (INDRASIRI & SIRIWARDENA 2018: 224). Dafür

wird eine Datei mit dem Namen *Dockerfile* angelegt. Die typischen Befehle, die in einem Dockerfile definiert werden, sind (vgl. WOLFF 2018: 69)

- FROM, um das Basis Docker Image zu spezifizieren,
- COPY/ADD, um Dateien aus dem Dateisystem hinzuzufügen,
- RUN, um Shell-Kommandos zu starten,
- ENTRYPOINT, um das Start-Skript im Container festzulegen und
- EXPOSE, um den Port des Containers nach außen freizugeben.

Docker Images können in eine Online-Registry geladen werden. Die Docker-Hub ist ein Open Source Projekt, das von Docker kostenlos bereitgestellt wird. Mit den Befehlen *docker pull* und *docker push* können Docker Images aus der Registry geladen bzw. hochgeladen werden (vgl. INDRASIRI & SIRIWARDENA 2018: 230).

### 4.4.2 Docker & Go

Das Basis-Image, auf dem alle Docker Images beruhen, wird *Scratch* genannt. Dies ist ein leeres Image, das weder von der DockerHub geladen noch in einem Docker Container gestartet werden kann. Go hat die Möglichkeit eine statisch gelinkte Binärdatei zu erstellen, die in einem solchen leeren Docker-Container laufen kann. Die Containergröße entspricht dann der Binärdatei.

### 4.4.3 Docker Compose

Docker Compose ist ein Tool von Docker, dass eine Multi-Container Entwicklungsumgebung definieren und managen kann. Es wird über eine YAML- (YAML Ain't Markup Language) Datei konfiguriert und kann dann über einen einzelnen Befehl gestartet werden (vgl. INDRASIRI & SIRIWARDENA 2018: 232f).

```
docker-compose up
```

Befehl 9: Starten einer Multi-Container Umgebung

Docker Compose sorgt nicht nur für das Starten der vielen Container, sondern kümmert sich auch um das Netzwerk, in dem sich die Container befinden. Es ist damit möglich, die Container unterschiedlichen Netzwerken zuzuordnen, um diesen die Kommunikation zu ermöglichen oder zu verbieten. (vgl. WOLFF 2018: 73)

# 5 Umsetzung

In diesem Kapitel wird das Konzept des Online-Formulars mithilfe der Technologien Angular, Docker, gRPC und Go umgesetzt. Es wird außerdem aufgezeigt, wie sich die in Kapitel 2 beschriebenen Anforderungen auf die Umsetzung auswirken.

Die Web-API, die zwischen dem Client und dem Webserver kommuniziert, ist als gRPC-Protobuf definiert und bildet die Formularlogik ab. Das gRPC-Verfahren koppelt den Client, der über eine Angular Library an die API angeschlossen wird, relativ stark an den Server, der über die Programmiersprache Go realisiert wird. Durch den Verzicht auf die Serialisierung und Deserialisierung von JSON (JavaScript Object Notation)-Objekten und die typsichere Kommunikation wird eine sehr sichere und trotzdem performante Web-API entworfen.

Die individuelle Lösung nutzt im Frontend das Material Angular Design. Aufgrund der streng definierten API ist es aber möglich, dieses Konzept auch über andere Frameworks und Designsprachen umzusetzen. Angular Material ist eine von den Angular-Erfindern entwickelte, hochwertige Benutzeroberfläche, die das zuvor bereits populäre Material Design adaptiert hat. Der Installationsprozess ist auf der Webseite beschrieben. (vgl. ANGULAR MATERIAL 2019)

Im Folgenden wird auf einzelne Komponenten in der gRPC-API eingegangen und die Anforderungen, die damit umgesetzt werden, erörtert. Dazu werden Teile aus dem Go-Package und der Angular Library vorgestellt, die die Schnittstelle implementieren und als modulare Bausteine in jeder Anwendung genutzt werden können.

```
syntax = "proto3";
package grpcform;
option go_package = "grpcform";

service FormService {
    rpc GetForm(GetFormRequest) returns (Form) {}
    rpc ValidateForm(Form) returns (Form) {}
    rpc SendForm(Form) returns (SendFormResponse) {}
}

message GetFormRequest {
    string name = 1;
}

message SendFormResponse {
    Form form = 1;
    bool succeed = 2;
    string message = 3;
}

message Form {
    string name = 1;
    repeated Field fields = 2;
    repeated Button buttons = 3;
    bool valid = 4;
}
```

Code 12: Methoden und Messages der gRPC-API (Anlage 2)

Der Code 12 zeigt das Grundgerüst der gRPC-API. Es werden die drei Methoden *GetForm* für die Initialisierung eines Formulars, *ValidateForm* für die serverseitige Validierung sowie *SendForm* für das Einfügen der Daten in eine Datenbank bereitgestellt. Zwischen dem Client und dem Server verkehren die *Form*-Messages, welche die zentrale Formularlogik darstellen. Im Initialisierungsprozess können mithilfe des *GetFormRequests* unterschiedliche Formulare über eine Namenskennung abgerufen werden. Die *SendFormResponse*-Message bietet zusätzlich zum *Form*-Objekt Informationen über den Erfolg der Eintragung in eine Datenbank an. Code 13 zeigt die Realisierung eines *GetForm*-Requests über TypeScript in der Angular Library. Diese verwaltet die *Form*-Class, welche für die Template-Modellierung genutzt wird. Die Interaktion mit den gekapselten Objekten erfolgt über *Getter*- und *Setter*-Methoden.

```
this.client = new FormServiceClient(this.host);
const req = new GetFormRequest();
req.setName(this.name);
this.client.getForm(req, null, (err, form: Form) => {
   if (err) {
      console.log(err);
      return;
   }
   this.form = form;
});
```

Code 13: Realisierung eines GetForm-Requests mit TypeScript (Anlage 5)

```
type FormServiceServer interface {
   GetForm(context.Context, *GetFormRequest) (*Form, error)
   ValidateForm(context.Context, *Form) (*Form, error)
   SendForm(context.Context, *Form) (*SendFormResponse, error)
}
```

Code 14: Interface zur Implementierung des gRPC-Servers in Go

Für die Implementierung eines Interfaces (Code 14) müssen die Methoden durch ein Go-*Type* bereitgestellt werden. Meist werden hierfür *Structs* verwendet. In diesem Fall wird die Datenstruktur *Map* genutzt, um unterschiedliche Formulare über einen Server liefern zu können. Zur Identifizierung wird der Name (siehe Code 12) eines *Form*-Objektes verwendet. Zu beachten ist, dass die Methoden, wenn sie von einem *Struct-Type* implementiert werden, meist als Pointer deklariert werden. Es kann verwirren, dass *Maps* in Go immer auf einen Speicher verweisen und kein Sternchen (*) zur Erkennung benötigen. Die Werte, die von dieser Datenstruktur gehalten werden, können dementsprechend innerhalb der Methoden verändert werden.

Als ersten Wert enthalten die Methoden einen sogenannten *Context* (Code 15). Dieses Interface wird von Google für das Weiterreichen von Werten und das Abbrechen von Signalen innerhalb eines Requests über die Grenzen von APIs hinweg genutzt. So können beispielsweise *Timeouts* für einen Request mitgegeben werden. Für den sinnvollen Einsatz sollten aber alle verteilten Systeme in einem Netzwerk diese Schnittstelle anbieten. (vgl. GOLANG 2014)

```
type ProxyServer map[string]element

func (ps ProxyServer) Start(host string) error {
   lis, _ := net.Listen("tcp", host)
   gs := grpc.NewServer()
   RegisterFormServiceServer(gs, ps)
   reflection.Register(gs)
   gs.Serve(lis)
   //
}

func (ps ProxyServer) GetForm(context.Context, *GetFormRequest) (*Form, error) {
   //
}

func (ps ProxyServer) ValidateForm(context.Context, *Form) (*Form, error) {
   //
}

func (ps ProxyServer) SendForm(context.Context, *Form) (*SendFormResponse, error) {
   //
}
```

Code 15: Implementierung des Server-Interfaces in Go (Anlage 10-13)

```
message Field {
    string label = 1;
    FieldStatus status = 2;
    bool instant_validate = 3;
    string error = 4;
    TextField text_field = 10;
    SelectField select_field = 11;
    NumericField numeric_field = 12;
    ActiveIf activeIf = 20;
    RequiredIf required_if = 21;
    DisabledIf disabled_if = 22;
    HiddenIf hidden_if = 23;
}

message Button {
    string label = 1;
    ButtonStatus status = 2;
    ButtonFuncType type = 3;
}
```

Code 16: Formularfelder und Buttons in der gRPC-API (Anlage 2/4)

Die Formularfelder und Buttons werden in dem Protocol Buffers Format der gRPC-API durch zahlreiche typisierte Eigenschaften charakterisiert (Code 16). Buttons haben z.B. eine Beschriftung, einen Status, mit dem ein Button deaktiviert oder verborgen werden kann, und einen *Type*, mit dem die Funktionen zum Zurücksetzten und Absenden eines Formulars spezifiziert werden. Ein Formularfeld bietet neben den Feldtypen, die hier *TextField*, *SelectField* und *NumericField* genannt werden, viele Möglichkeiten an, mit denen die Formularfelder validiert werden können. Über die Eigenschaften *ActiveIf*, *RequiredIf*, *DisabledIf* und *HiddenIf* kann der Status eines Formularfeldes anhand der Werte anderer Felder verändert werden. In Code 17 wird die Template-Syntax in der Angular App dargestellt. Über das Property-Binding können die Attribute *hidden* (*div*), *disabled* und *required* (*input*) mittels der Berechnung von Wahrheitswerten verändert werden. Dem Status-Type liegt eine *Enumeration*, also ein Aufzählungstyp zugrunde, der für jede Implementierung des Typs einen Zahlenwert hinterlegt (Bsp. 4 für *FIELD_STATUS_HIDDEN*).

```
<div *ngFor="let field of form.getFieldsList(); index as i"
     [hidden]="field.getStatus() == FIELD_STATUS.FIELD_STATUS_HIDDEN">
    <div *ngIf="field.getTextField(); let textField">
        <mat-form-field (change)="textField.setValue($event.target.value);
            validateField(field)">
            <input matInput [placeholder]="field.getLabel()"
                   [value]="textField.getValue()" [matAutocomplete]="auto"
                   [attr.disabled]="field.getStatus() ==
                        FIELD_STATUS.FIELD_STATUS_DISABLED"
                   [required]="field.getStatus() ==
                        FIELD_STATUS.FIELD_STATUS_REQUIRED">
```

Code 17: Template Syntax der Angular Library (Anlage 7)

```
message HiddenIf {
    repeated Validator validators = 1;
}

message Validator {
    int64 index = 1;
    string text_is_equal = 10;
    int64 length_smaller_than = 11;
    int64 length_greater_than = 12;
    int64 number_is_equal = 20;
    int64 number_smaller_than = 21;
    int64 number_greater_than = 22;
    string match_regex_pattern = 30;
}
```

Code 18: Validator in der gRPC-API (Anlage 2/3)

Validatoren überprüfen einen Wert anhand mathematischer Operatoren, der Textlänge, der Wertgleichheit sowie über reguläre Ausdrücke und geben einen Wahrheitswert zurück. Mit regulären Ausdrücken können bestimmte Zeichen, Buchstaben oder Zahlen erlaubt bzw. verboten werden (Code 21). Für die Formularvalidierung ist der reguläre Ausdruck die flexibelste Möglichkeit die Eingaben zu untersuchen, da diese sämtliche anderen Validierungen erzeugen können. Aus Performanceaspekten wurden diese aber extra implementiert. Nicht alle Eigenschaften der Validatoren können auf jedes Formularfeld angewendet werden. Code 19 zeigt einen Go-Ausschnitt in dem das *TextField* serverseitig auf Textgleichheit, Textlänge und auf reguläre Ausdrucke (*regex* – Go Standard Bibliothek) untersucht wird.

```
func checkValidatorOnTextField(
   textField *TextField,
   validator *Validator,
   ) bool {
   if v := validator.GetTextIsEqual(); v != "" &&
      textField.GetValue() == v {
      return true
   }
   if v := validator.GetLengthSmallerThan(); v != 0 &&
      int64(len(textField.GetValue())) < v {
      return true
   }
   if v := validator.GetLengthGreaterThan(); v != 0 &&
      int64(len(textField.GetValue())) > v {
      return true
   }
   if v := validator.GetMatchRegexPattern(); v != "" {
      if ok, err := regexp.MatchString(v, textField.GetValue()); ok &&
         err != nil {
         return true
      }
   }
   return false
}
```

Code 19: Serverseitige Validierung eines Textfeldes in Go (Anlage 13)

```
message TextField {
    string value = 1;
    repeated Option options = 2;
    int64 min = 10;
    string min_error = 20;
    int64 max = 11;
    string max_error = 21;
    string regex = 12;
    string regex_error = 22;
}

message Option {
    int64 index = 1;
    string value = 2;
}
```

Code 20: TextField in der gRPC-API (Anlage 3)


Abbildung 15: Input/Autocomplete (ANGULAR MATERIAL 2019)

Das *TextField* repräsentiert das Eingabefeld mit Vorschlägen aus dem Kapitel 2.1.1. Neben der freien Eingabe von beliebigen Werten können über *Options* Vorschläge an das Frontend übergeben werden (Code 20). Das Angular Material Framework bietet hierfür eine *Input*-Komponente an, die mit einer *Autocomplete*-Komponente erweitert werden kann. Bei einem Klick in das Feld werden die Vorschläge direkt angezeigt (Abb. 15).

```
USERNAME: {
   InstantValidate: true,
   Label:           "Please enter your username",
   Status:          grpcform.FieldStatus_FIELD_STATUS_REQUIRED,
   TextField: &grpcform.TextField{
      Min:        5,
      MinError:   "Too Short",
      Max:        25,
      MaxError:   "Too long",
      Regex:      "^[a-zA-Z]{5,25}$",
      RegexError: "Only letters",
   },
},
```

Code 21: Erzeugen eines TextFields in Go (Anlage 14)

Daneben bietet das *TextField* zahlreiche Eigenschaften an, um dem Nutzer Hilfestellungen bei einer falschen Eingabe zu geben. Die freie Eingabe von Textwerten ist eine große Fehlerquelle in Formularen, der durch eine sinnvolle Fehlerbehandlung entgegengewirkt werden kann. *Min* und *Max* geben hierbei die erlaubten Textlängen an. Mit dem regulären Ausdruck in Code 21 werden nur Texte erlaubt, die aus 5 bis 25 Buchstaben bestehen. Die Werte für *Min* und *Max* sind für die Geschäftslogik daher nicht relevant. Der Vorteil besteht neben dem Performancegewinn darin, dass individuelle Fehlerbenachrichtigungen ausgegeben werden können.

```
message SelectField {
    int64 index = 1;
    SelectType type = 2;
    repeated Option options = 3;
    string error = 4;
}

enum SelectType {
    SELECT_TYPE_UNSPECIFIED = 0;
    SELECT_TYPE_SIMPLE = 1;
    SELECT_TYPE_MULTI = 2;
}
```

Code 22: Auswahlfelder spezifiziert in der gRPC-API (Anlage 3)

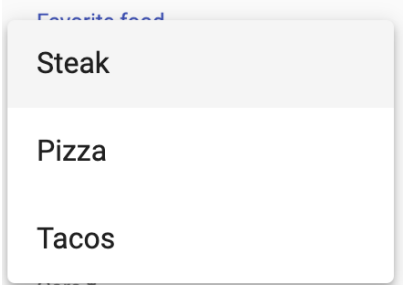


Abbildung 16: Select (ANGULAR MATERIAL 2019)

Auswahlfelder können auf unterschiedlichste Weisen dargestellt werden. In diesem Fall werden die Komponenten *Radio Group* und *Select* von Material Angular verwendet. In der Abbildung 16 ist die Select-Komponente dargestellt. Diese erhält in der *SelectField*-Message den Type *SELECT_TYPE_MULTI*. *Radio Buttons* wie in Code 23 werden über den *Enumerations*-Wert *SELECT_TYPE_*SIMPLE definiert. Standardwerte werden in der RPC-Welt oft nicht überliefert, um die Größe der Nachrichten zu reduzieren. Daher sollten die Nullwerte wie die des Aufzählungstyps *SelectTypes* mit *UNSPECIFIED* belegt werden (Code 22).

Die Optionen werden im Falle der *Radio Group* durch *Radio Buttons* erzeugt. Die *ngFor*-Direktive iteriert über das *Option Array* und erzeugt somit die DOM-Elemente (Code 23). Die Werte der Felder werden numerisch im Index gespeichert. Über das Event-Binding (*change*) kann der Index des *SelectFields* mit eine *Setter*-Methode gespeichert werden.

```
<div *ngIf="select.getType() == SELECT_TYPE.SELECT_TYPE_SIMPLE">
    <div *ngIf="field.getLabel()">
        <label>{{ field.getLabel() }}<br></label>
    </div>
    <mat-radio-group (change)="select.setIndex($event.value);
        validateField(field)" [value]="select.getIndex()"
         [disabled]="field.getStatus() == FIELD_STATUS.FIELD_STATUS_DISABLED"
         [required]="field.getStatus() == FIELD_STATUS.FIELD_STATUS_REQUIRED">
        <mat-radio-button *ngFor="let option of select.getOptionsList()"
                          [value]="option.getIndex()">{{ option.getValue() }}
        </mat-radio-button>
    </mat-radio-group>
</div>
```

Code 23: Angular Template-Syntax für ein simples Auswahlfeld (Anlage 7)

```
CAR: {
   Label:           "Do you have a car?",
   InstantValidate: true,
   Status:          grpcform.FieldStatus_FIELD_STATUS_REQUIRED,
   SelectField: &grpcform.SelectField{
      Index: NO,
      Type:  grpcform.SelectType_SELECT_TYPE_SIMPLE,
      Options: []*grpcform.Option{
         {Index: NO, Value: "No"},
         {Index: YES, Value: "Yes"},
      },
   },
   HiddenIf: &grpcform.HiddenIf{
      Validators: []*grpcform.Validator{
         {
            Index:             AGE,
            NumberSmallerThan: 18,
         },
      },
   },
},
```

Code 24: Implementierung eines Select-Feldes in Go (Anlage 15)

Die Protobuf-Definition in Code 22 bietet zwei Arten von Select-Feldern an, die von den Frontend-Implementierungen unterschiedlich umgesetzt werden können. Das Beispiel in Code 24 erzeugt einen *Radio Button* in der Angular Material Komponente (Code 23). Wie in Kapitel 2.1.1 beschrieben, zeichnet sich dieser Typ meist durch Fragen aus, die mit Ja und Nein beantwortet werden können. *InstantValidate* ist eine Eigenschaft, die dem Frontend mitteilt, dass dieses Feld bei jeder Änderung direkt durch das Backend validiert werden soll. Der *HiddenIf*-Validator setzt den Status dieses *CAR*-Feldes auf *FIELD_STATUS_HIDDEN*, sobald das Feld AGE einen Wert kleiner als 18 besitzt. Minderjährige Nutzer werden dementsprechend nicht mehr nach dem Besitz eines Autos befragt.

Die Abbildung 17 zeigt einen *Slider* aus dem Angular Material Design. Mit diesem werden die numerischen Felder aus der Protobuf-Definition in Code 25 modelliert. Sie erlauben die Darstellung einer Zahlenskala mit einem Minimal- und Maximalwert sowie einer Stufenbreite.

```
message NumericField {
    int64 value = 1;
    int64 step = 2;
    int64 min = 10;
    string min_error = 20;
    int64 max = 11;
    string max_error = 21;
}
```

Code 25: Numerische Felder in der gRPC-API (Anlage 3)

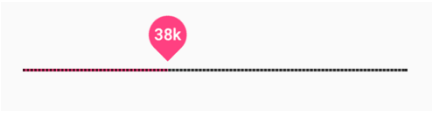


Abbildung 17: Slider (ANGULAR MATERIAL 2019)

```
<div *ngIf="field.getNumericField(); let numericField">
    <div *ngIf="field.getLabel()">
        <label>{{ field.getLabel() }}<br></label>
    </div>
    <mat-slider (change)="numericField.setValue($event.value);
        validateField(field)" thumbLabel
        [step]="numericField.getStep()" [min]="numericField.getMin()"
        [max]="numericField.getMax()" [value]="numericField.getValue()"
        [disabled]="field.getStatus() == FIELD_STATUS.FIELD_STATUS_DISABLED">
    </mat-slider>
</div>
```

Code 26: Numerisches Feld als Angular Material Slider (Anlage 8)

Der *Slider* von Material Angular kann die meisten Attribute eines numerischen Feldes implementieren (Code 26). Eine Fehlerbehandlung ist in diesem Fall nicht notwendig. Die Werte werden als *Number*-Datentyp von TypeScript gespeichert und über gRPC als *Int64*-Werte an den Server gesendet. Eine Validierung dieser Felder ist meist nicht notwendig, wird aber serverseitig zur Verhinderung von Serverangriffen durchgeführt (Code 27).

Code 27 zeigt einen Ausschnitt aus der *ValidateForm*-Methode des gRPC-Servers in Go. Dabei werden die eingehenden Werte aufgrund bestimmter Bedingungen in ein neues *Form*-Objekt geschrieben. Beispielsweise ist eine Bedingung, dass Felder aktiv oder benötigt (*required*) sein müssen. Dieser Status wird zuvor über die Validierungsmethoden in Code 19 ermittelt. Der Aufbau verhindert die Korruption des Formulars und des Servers von außen und stellt die Formularlogik in jedem Fall sicher.

```
if inNumericField := inField.GetNumericField(); hasStatus(outField.GetStatus(),
   FieldStatus_FIELD_STATUS_ACTIVE, FieldStatus_FIELD_STATUS_REQUIRED) &&
   inNumericField != nil {
   outSlider := outField.GetNumericField()
   outSlider.Value = inNumericField.GetValue()
   if !out.Valid || (outField.GetStatus() == FieldStatus_FIELD_STATUS_ACTIVE &&
      outSlider.Value == 0) {
      continue
   }
   v := outSlider.GetValue()
   if int64(v) < outSlider.GetMin() {
      outField.Error = outSlider.GetMinError()
      out.Valid = false
      continue
   }
   if int64(v) > outSlider.GetMax() {
      outField.Error = outSlider.GetMaxError()
      out.Valid = false
      continue
   }
}
```

Code 27: Validierung eines numerischer Felder in Go (Anlage 12)

```
<ng-mat-grpc-form name="car" host="http://localhost:8080"
                  (success)="onSuccess($event)"></ng-mat-grpc-form>
```

Code 28: Benutzen der Angular Library in einem Projekt

Die Angular Library, die mit Angular Material realisiert wurde, kann in die NPM-Online-Registry hochgeladen und anschließend in jedem Angular Projekt genutzt werden. Code 28 zeigt ein Beispiel für die Nutzung dieser Bibliothek. Nachdem die Library installiert wurde, kann die Angular Component dieser Bibliothek per *Selector* (Code 29: *ng-mat-grpc-form*) in die App integriert werden. Der Komponente wird der Name des Formulars und die Adresse des gRPC-Servers übergeben, von dem das Formular automatisch geladen wird. Nach erfolgreicher Bearbeitung wird über das Event-Binding die Variable *success* emittiert, und dadurch in der App die Methode *onSuccess* aufgerufen.

```
@Component({
   selector: 'ng-mat-grpc-form',
   template: `…`,
   styles: []
})
export class NgMatGrpcFormComponent implements OnInit {
   @Input() name: string;
   @Input() host: string;
   @Output() success = new EventEmitter<string>();
   …
```

Code 29: Input und Output Binding in der Angular Component (Anlage 5)

Die Abbildung 18 zeigt als Ergebnis die unterschiedlichen Bearbeitungsstufen des in dieser Arbeit genutzten Formulars. Auf der linken Seite ist das initialisierte Formular zu sehen. Die beiden anderen Abbildungen zeigen die Fehlerbehandlung sowie die Aktivierung und Deaktivierung von Formularelementen.

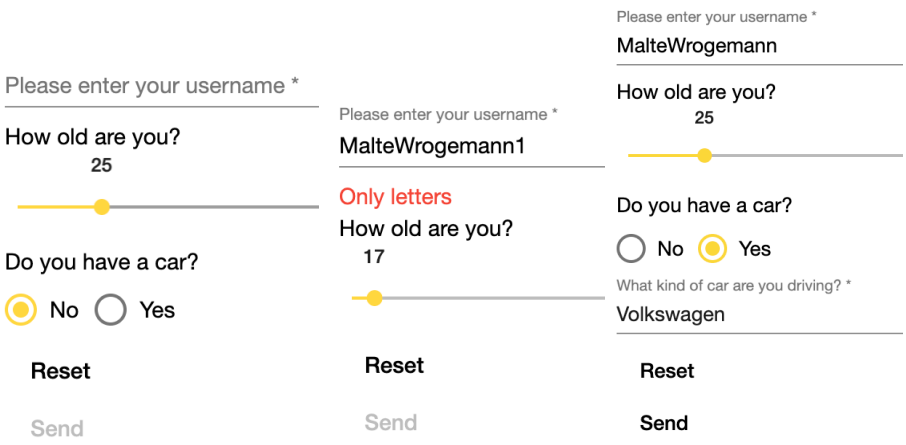


Abbildung 18: Das erstellte Formular (eigene Darstellung)

```yaml
database:
  image: postgres:10.1
  env_file:
    - db.env
  volumes:
    - ./backend/database:/docker-entrypoint-initdb.d/
  ports:
    - 5432:5432
database-service:
  build:
    context: .
    dockerfile: database-service.Dockerfile
  env_file:
    - db.env
  ports:
    - 9000:9000
grpc-form-service:
  build:
    context: .
    dockerfile: service.Dockerfile
  ports:
    - 50051:50051
frontend:
  build: ./frontend
  depends_on:
    - grpc-form-service
    - proxy
  ports:
    - 80:80
  volumes:
    - ./frontend:/usr/src/app:rw
proxy:
  image: envoyproxy/envoy-alpine:v1.10.0
  environment:
    NODE_ENV: development
  ports:
    - 8080:8080
    - 8081:8081
  volumes:
    - ./envoy.yaml:/etc/envoy/envoy.yaml
```

Code 30: Entwicklungsumgebung mit Docker Compose

Die Entwicklungsumgebung eines solchen Projekts kann mit Docker Compose aufgesetzt werden (Code 30). Zusätzlich zu den in diesem Kapitel bereits beschriebenen Frontend- und Backend- (*grpc-form-service*) Containern, wird ein Envoy Proxy (Kap. 4.1.2) sowie eine Datenbank benötigt, auf die über einen Datenbank-Service, der ebenfalls in Go implementiert ist, zugegriffen werden kann. Alle Container befinden sich in dem Standard-Netzwerk und können miteinander über ihre Bezeichnung und die freigegebenen Ports kommunizieren. In *db.env* werden die Umgebungsvariablen in die Container *database* und *database-service* übergeben. Das *postgres*-Image erzeugt daraus automatisch eine Datenbank, auf die der *databank-service* über die Umgebungsvariablen zugreifen kann. In den Dockerfiles der beiden Services werden Linux-Binaries erzeugt, die in einem nahezu leeren Container gestartet werden.

```
service Database {
    rpc GetUser(GetUserRequest) returns (User) {}
    rpc InsertUser(User) returns (User) {}
}

message GetUserRequest {
    string name = 1;
}

message User {
    string name = 1;
    int64 age = 2;
    string car = 3;
}
```

Code 31: Kommunikation zwischen den Backend-Services mit gRPC

Die Kommunikation zwischen dem Datenbank-Service und der Formularlogik in dem *grpc-form-service* findet ebenfalls über gRPC statt (Code 31). Der Datenbank-Service agiert hierbei als gRPC-Server, der Anfragen vom *grpc-form-service* erhält. Der gRPC-Client kann den Datenbank-Service über die Bezeichnung und den freigegebenen Port in der Docker Compose Konfigurationsdatei erreichen (Code 32). Die in Code 31 spezifizierten Methoden können dann an den *DatabaseClient* übergeben werden.

```
conn, _ := grpc.Dial("database-service:9000", grpc.WithInsecure())
db := api.NewDatabaseClient(conn)
u, err := db.InsertUser(ctx, &api.User{
   Name: out.GetFields()[USERNAME].GetTextField().GetValue(),
   Age:  out.GetFields()[AGE].GetNumericField().GetValue(),
   Car:  car,
})
```

Code 32: gRPC-Client in Go

Code 33 zeigt die SQL (Structured Query Language) -Befehle, die für den Initialisierungsprozess der Datenbank in der Konfigurationsdatei (Code 32) in das *docker-entrypoint-init.db*-Verzeichnis hinzugefügt werden. Die eingetragenen, neuen Werte werden ebenfalls in diesem Verzeichnis gespeichert und stehen daher auch bei einem Neustart der Entwicklungsumgebung wieder zur Verfügung.

```
CREATE TABLE usercars (
    username varchar(50),
    age INT,
    car varchar(50)
);

INSERT INTO usercars (username, age, car)
VALUES
('Malte', 25, 'Volkswagen');
```

Code 33: Datenbank-Initialisierung

```
import (
   _ "github.com/lib/pq"
   "database/sql"
   …
)

func main() {
   database := os.Getenv("POSTGRES_DB")
   user := os.Getenv("POSTGRES_USER")
   password := os.Getenv("POSTGRES_PASSWORD")
   connStr := fmt.Sprintf("host=database port=5432 user=%s dbname=%s password=%s
sslmode=disable", user, database, password)
   db, _ := sql.Open("postgres", connStr)
   defer db.Close()
   lis, _ := net.Listen("tcp", ":9000")
   s := grpc.NewServer()
   api.RegisterDatabaseServer(s, &server{db: db})
   reflection.Register(s)
   s.Serve(lis)
}

func (s *server) GetUser(ctx context.Context, req *api.GetUserRequest) (*api.User,
error) {
   var username string
   var age int
   var car string
   row := s.db.QueryRow("SELECT username, age, car FROM usercars WHERE
username=$1", req.GetName())
   row.Scan(&username, &age, &car)
   return &api.User{Name: username, Age: int64(age), Car: car}, nil
}
```

Code 34: Ausschnitt aus dem Database-Service

Der Datenbank-Service agiert, wie in Code 34 zu sehen, als gRPC-Server, der auf *Port 9000* Requests eines gRPC-Clients erwartet. Das *server-Struct* implementiert die Schnittstelle und enthält eine Verbindung zur PostgreSQL-Datenbank. Es wird ein SQL-*Driver* genutzt, der von der Standardbibliothek *database/sql* verarbeitet werden kann. Es reicht hierbei aus, die Bibliothek (LIB/PQ 2019) über einen Unterstrich zu importieren. Dabei wird ein Initialisierungsprozess gestartet, der den *Driver* registriert.

Das in diesem Kapitel umgesetzte Beispiel ermöglicht die einfache Implementierung eines Formulars in Angular Applikationen über die Nutzung einer Angular Library und einem Go Package. Im Frontend wird mit einem HTML-Tag auf einen gRPC-Server verwiesen, der das Formular modelliert und validiert. Im zweiten Schritt kann der vorgestellte Datenbank-Service als eine lose gekoppelte Komponente im Sinne der Microservices-Architektur genutzt werden.

# 6 Bewertung

Ziel dieser Arbeit war es, ein Konzept für die systematische Erstellung von Online-Formularen in modernen Webanwendungen zu gestalten und in einem konkreten Projekt mit den Technologien Angular, Docker, gRPC und Go umzusetzen. Dafür wurden in Kapitel 2 Anforderungen entwickelt, die nun mit den Ergebnissen verglichen werden.

Das Konzept dieser Arbeit basiert auf einer gRPC-API, welche die Formularlogik strikt vorgibt. Komponenten, die diese API konsumieren bzw. implementieren, können frei (auch in ihrer Technologiewahl) ausgetauscht werden. Allerdings ist es notwendig, dass sämtliche gRPC-Clients und -Server ihren Funktionsumfang genau beschreiben.

Die Formularlogik liegt hier klar bei der Server-Komponente. Während serverseitig sämtliche Formularlogik abläuft, ist das Frontend alleine für die Darstellung des *Form*-Objektes zuständig. Dabei können unterschiedliche Frameworks und Design Sprachen verwendet werden. Für die Darstellung, der in der API genutzten Feldern wie den Eingabefeldern mit Vorschlägen, Auswahlfeldern und numerischen Feldern, sollte aber jede Technologie in diesem Bereich Lösungen anbieten.

Bei der serverseitigen Validierung wird das gesamte Form-Objekt zwischen Client und Server ausgetauscht. Sämtliche Felder und Buttons können so beliebig verändert werden. Zudem ist es möglich, clientseitige Lösungen zu erlauben, wenn das Attribut *InstantValidate* eines Feldes ausgeschaltet wird oder keine ausreichende Internetverbindung besteht. Die Validierungen können wie in der Beispiel Go API (Anlage 14-17) auch clientseitig mit TypeScript realisiert werden. (Anlage 6)

Reguläre Ausdrücke erlauben es sämtliche Arten von Eingaben zu untersuchen. Daher ist es möglich dem Nutzer in jeder Situation hilfreiche Fehlermeldungen zu bieten.

Die erarbeitete Lösung, bestehend aus einer Angular Library und einem Go Package, vereinfacht das Aufsetzen von kleinen und komplexen Formularen. Es ist nicht mehr nötig, die Darstellung an die Logik anzupassen. Stattdessen wird nun das Formular serverseitig modelliert und über eine Namenskennung bereitgestellt. Frontend-Lösungen wie die Angular Library, die auf Angular Material Komponenten beruht, können diese Logik dann in eine Darstellung übersetzten.

Die Entwicklungsumgebung mit Docker und Docker Compose vereinfacht die Umsetzung komplexer Softwarearchitekturen. Gerade die Möglichkeit einen Proxy-Server wie Envoy oder eine Datenbank wie PostgreSQL über einen Container lokal in wenigen Sekunden starten zu können, beschleunigt die Softwareentwicklung enorm. Zudem eignet sich Docker ideal für die Verpackung von Microservices und die Nutzung dieser in Cloud-Architekturen.

Go ist eine Programmiersprache, die sich aufgrund der Möglichkeit, statisch gelinkte Binärdateien für unterschiedlichste Betriebssysteme zu erzeugen, ideal für die Nutzung mit Docker eignet. Außerdem bietet Go eine klare und aufgeräumte Syntax sowie eine große Standardbibliothek, die für die Entwicklung von Webapplikationen und Microservices viele Vorteile bietet (z.B. Goroutines und Channels). Trotz der automatischen Speicherbereinigung, sowie der direkten Kompilierung in Maschinencode ist die Kompilierungszeit von Go-Programmen zudem recht performant.

Alle genutzten Technologien haben sich in diesem Projekt bewährt und können in der Zukunft vom Landesamt für Geoinformation und Landesvermessung Niedersachsen für die Entwicklung moderner Webapplikationen genutzt werden.

# 7 Zusammenfassung

Zu Beginn der Arbeit wurden aus einem analogen Fragebogen der Gutachterausschüsse für Grundstückswerte die Anforderungen ermittelt, die für ein Online-Formular relevant sind. Daraufhin wurde ein Konzept entwickelt, das auf der Verknüpfung der Anforderungen mit den Technologien Angular, Docker, gRPC und Go beruht. Basierend auf einer Einführung in diese vier Themenbereiche, wurde dann ein solches Formular umgesetzt.

Wie in der Bewertung in Kapitel 6 beschrieben, wurden sämtliche Anforderungen in einer Angular Library und einem Go Package derart umgesetzt, dass Online-Formulare nun wie Bausteine in anderen Webapplikationen implementiert werden können. Das Konzept der gRPC-Web-API ist in der Hinsicht universell, als dass es auch mit anderen Programmiersprachen bzw. Frameworks einfach umgesetzt werden kann.

Gerade gRPC für die Nutzung im Web befindet sich noch in der Anfangszeit. Daher gibt es nur wenige Quellen oder Beispielprojekte, die ein derartiges Projekt zum Zeitpunkt dieser Arbeit umgesetzt haben. Aber wie sich herausgestellt hat, lohnt es sich diese Technologie weiter zu verfolgen.

Die Vorteile, die Docker bei der Entwicklung komplexer Softwarearchitekturen bietet, sind unbestritten. Zudem hat sich in diesem Projekt gezeigt, dass die Programmiersprache Go insbesondere im Zusammenhang mit Docker sehr gut in den Bereichen Microservices und Cloud Computing eingesetzt werden kann.

In der schnelllebigen JavaScript-Welt hat sich Angular als eine konstante Größe unter den Frameworks etabliert und auch in dieser Arbeit seine Vorteile bewiesen. Zur Bewertung dieser Technologie sollte das Online-Formular aber auch mit den JavaScript-Frameworks React.js und Vue.js umgesetzt werden.

# Literatur


ALEXSOFT (2019): The Good and the Bad of Angular Development. https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/. Stand 24.07.2019. Zuletzt abgerufen am 10.07.2019.

ANGULAR (2019): Creating Libraries. https://angular.io/guide/creating-libraries. Zuletzt abgerufen am 07.07.2019.

ANGULAR MATERIAL (2019): Angular Material. https://material.angular.io/. Zuletzt abgerufen am 21.07.2019.

BALZERT H. (2011): Lehrbuch der Softwaretechnik. Entwurf, Implementierung, Installation und Betrieb. 3., Auflage. Springer-Verlag. Berlin. Heidelberg.

BEZ R. (2014): CSS-Präprozessoren im Vergleich. https://www.heise.de/developer/artikel/CSS-Praeprozessoren-im-Vergleich-2288284.html. Stand 08.08.2014. Zuletzt abgerufen am 20.07.2019.

BITKOM (2015): Entwicklung erfolgreicher Webanwendungen. Leitfaden Webentwicklung 2015. https://www.bitkom.org/sites/default/files/pdf/noindex/Publikationen/2015/Leitfaden/Entwicklung-erfolgreicher-Webanwendungen/LF-Webanwendungen-150910-1.pdf. Stand 10.09.2015. Zuletzt abgerufen am 05.07.2019.

BLOCH J. (2014): A Brief, Opinionated History of the API. http://goo.gl/WLKLQc. Stand 21.10.2014. Zuletzt abgerufen am 22.07.2019.

CHENEY D. (2017): Understand Go pointers in less than 800 words or your money back. https://dave.cheney.net/2017/04/26/understand-go-pointers-in-less-than-800-words-or-your-money-back. Stand 26.04.2017. Zuletzt abgerufen am 10.07.2019.

DOCKER (2019): Comparing Docker Containers and Virtual Maschines. https://www.docker.com/resources/what-container. Zuletzt abgerufen am 14.07.2019.

DUNKEL J., A. HOLITSCHKE (2003): Softwarearchitektur für die Praxis. 1., Auflage. Springer-Verlag. Berlin. Heidelberg.

GAG-NIEDERSACHSEN (2019): Kaufpreissammlung. https://www.gag.niedersachsen.de/gutachterausschuesse/kaufpreissammlung/kaufpreissammlung-88079.html. Zuletzt abgerufen am 12.07.2019.

GERRAND A. (2012): Go and the Zen of Python. https://talks.Golang/2012/zen.slide. Zuletzt abgerufen am 10.07.2019.

GOLANG (2014): Go Concurrency Patterns. Context. Stand 29.07.2014. https://blog.golang.org/context. Zuletzt abgerufen am 21.07.2019.

GOLANG (2019a): Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. https://golang.org/. Zuletzt abgerufen am 10.07.2019.

GOLANG (2019b): Packages. https://golang.org/pkg/ Zuletzt abgerufen am 10.07.2019.

GOLANG (2019c): Package io. https://golang.org/pkg/io Zuletzt abgerufen am 10.07.2019.

GOLANG (2019d): Package io/ioutil. https://golang.org/pkg/io/ioutil Zuletzt abgerufen am 10.07.2019.

GOLANG (2019e): Package net/http. https://golang.org/pkg/net/http/. Zuletzt abgerufen am 10.07.2019.

GOLANG (2019f): Results for query Read. https://golang.org/search?q=Read#Global. Zuletzt abgerufen am 10.07.2019.

GOLANG (2019g): Go Command. https://golang.org/cmd/go/. Zuletzt abgerufen am 10.07.2019.

GOOGLE (2019): Developer Guide. https://developers.google.com/protocol-buffers/docs/overview. Zuletzt abgerufen am 15.07.2019.

GOWITEK (2018): Companies Using Golang. https://www.gowitek.com/golang/blog/companies-using-golang. Stand 31.12.2018. Zuletzt abgerufen am 15.07.2019.

GRPC (2019): Documentation Guides. https://grpc.io/docs/guides/. Zuletzt abgerufen am 15.07.2019.

IMPROPABLE-ENG (2019): gRPC-Web. Typed Frontend Development. https://github.com/improbable-eng/grpc-web. Zuletzt abgerufen am 07.07.2019. Improbable-Engineering.

INDRASIRI K, SIRIWARDENA P. (2018): Microservices of the Enterprise. Designing, Developing, and Deploying. Apress Media LCC. New York.

LACKES R. & SIEPERMANN M. (2018): Compiler. https://wirtschaftslexikon.gabler.de/definition/compiler-30434/version-254016. Stand 19.02.2018. Zuletzt aufgerufen am 20.07.2019.

LACKES R. & SIEPERMANN M. (2018): Open Source. https://wirtschaftslexikon.gabler.de/definition/open-source-43032/version-266368. Stand 19.02.2018. Zuletzt aufgerufen am 20.07.2019.

LIB/PQ (2019): pq - A pure Go postgres driver for Go's database/sql package. https://github.com/lib/pq. Zuletzt abgerufen am 21.07.2019.

MALCHER F., HOPPE J. & KOPPENHAGEN D. (2019): Angular. Grundlagen, fortgeschrittene Themen und Best Practices. 2., aktualisierte und erweiterte Auflage. dpunkt.verlag. Heidelberg.

MENGE-SONNENTAG R. (2016): Erste Beta von Googles gRPC verfügbar. https://www.heise.de/developer/meldung/Erste-Beta-von-Googles-gRPC-verfuegbar-2862634.html. Stand 28.10.2016. Zuletzt abgerufen am 10.07.2019.

MINHAS S. (2008): 7 Rules of Using Radio Buttons vs Drop-Down Menus. https://blog.prototypr.io/7-rules-of-using-radio-buttons-vs-drop-down-menus-fddf50d312d1. Stand 06.05.2018. Zuletzt abgerufen am 06.07.2019.

NODE.JS (2019): Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. https://nodejs.org/en/. Zuletzt abgerufen am 07.07.2019.

NOBLE N. (2016): gRPC Releases. https://github.com/grpc/grpc/releases?after=v1.2.0-pre2. Stand 19.08.2016. Zuletzt abgerufen am 10.07.2019.

ONLINEMARKETING-PRAXIS (2019): Mobile First. https://www.onlinemarketing-praxis.de/glossar/mobile-first. Zuletzt abgerufen am 20.07.2019.

PATEL K. (2017): Why should you learn Go? https://medium.com/@kevalpatel2106/why-should-you-learn-go-f607681fad65. Stand 08.01.2017. Zuletzt abgerufen am 07.07.2019.

PERKINS L., S. CHEUNG & K.SETHURAMAN (2018a): gRPC-Web is Generally Available. https://grpc.io/blog/grpc-web-ga/. Stand 23.10.2018. Zuletzt abgerufen am 10.07.2019.

PERKINS L. (2018b): Envoy and gRPC-Web. a fresh alternative to REST. https://blog.envoyproxy.io/envoy-and-grpc-web-a-fresh-new-alternative-to-rest-6504ce7eb880. Stand 01.11.2018. Zuletzt abgerufen am 10.07.2019.

PIKE R. (2013): Concurrency is not Parallelism. https://blog.Golang/concurrency-is-not-parallelism. Stand 16.01.2013. Zuletzt abgerufen am 10.07.2019.

REDDY M. (2011): API Design for C++. 1., Edition. Morgan Kaufmann. Burlington.

RODEN G. (2008): Auf der Fährte von C#. 1., Auflage. Springer-Verlag. Berlin. Heidelberg.

ROUSE M. (2014): Document Object Model. https://www.computerweekly.com/de/definition/Document-Object-Model-DOM. Stand 04.2014. Zuletzt abgerufen am 20.07.2019.

SANDOVAL K. (2018): When to Use What. REST, GraphQL, Webhooks, & gRPC. https://nordicapis.com/when-to-use-what-rest-graphql-webhooks-grpc/. Stand 15.07.2018. Zuletzt abgerufen am 04.07.2019.

SIEPERMANN M. (2018): Single Page Application. https://wirtschaftslexikon.gabler.de/definition/single-page-application-54485/version-277514. Stand 19.02.2018. Zuletzt aufgerufen am 20.07.2019.

SPICHALE K. (2017): API-Design. Praxishandbuch für Java- und Webservice-Entwickler. Korrigierter Nachdruck 2017. dpunkt.verlag. Heidelberg.

STACKOVERFLOW (2019): StackOverflow Trends. Angular, React.js, Vue.js. https://insights.stackoverflow.com/trends. Zuletzt abgerufen am 15.07.2019.

TAYLOR I. L. (2009): Go Interfaces. https://www.airs.com/blog/archives/277. Stand 27.11.2009. Zuletzt abgerufen am 10.07.2019.

TECHNOPEDIA (2019a): Command Line Interface. https://www.techopedia.com/definition/3337/command-line-interface-cli. Zuletzt abgerufen am 20.07.2019.

TECHNOPEDIA (2019b): Command Line Interface. https://www.techopedia.com/definition/12806/garbage-collector. Zuletzt abgerufen am 20.07.2019.

TILKOV S., EIGENBRODT M., SCHREIER S. & WOLF O. (2015): REST und HTTP. Entwicklung und Integration nach dem Architekturstil des Web. 3., aktualisierte und erweiterte Auflage. dpunkt-verlag. Heidelberg.

WERNER C. (2011): Verteilte Systeme. https://www.ibr.cs.tu-bs.de/courses/ss11/vs/VS-Kap01-Einfuehrung-Single.pdf. Zuletzt abgerufen am 20.07.2019.

WOLFF E. (2018): Das Microservices-Praxisbuch. Grundlagen, Konzepte und Rezepte. 1., Auflage. dpunkt.verlag. Heidelberg.

# Befehle



# Code

# Anlagen

## Anlage 1: Fragebogen für Ein- und Zweifamilienhäuser

**GAG** Gutachterausschuss für Grundstückswerte
#FBOG_GANAM#

#FBOG_GASTR# #FBOG_GAPLZ# #FBOG_GAORT#
Tel. #FBOG_GATEL# Fax #FBOG_GAFAX#

| Bearbeiter: | Kauffall-Nr.: |
|---|---|
| #FBOG_BEARB# | #FBOG_KFKZ# |

### Fragebogen für Ein- und Zweifamilienhäuser

Wir bitten Sie, Ihre Angaben auf den **Zeitpunkt des Erwerbs** zu beziehen.
Zutreffendes bitte ankreuzen ☒ bzw. ausfüllen ⌊⌋.
Die eingeklammerten Zahlen sind interne Verschlüsselungen.

Lage: #FBOG_LABE# #FBOG_HAUS# #FBOG_PLZG# #FBOG_GEDE# #FBOG_GEMA#
(Straße, Haus-Nr.) (PLZ) (Stadt/Gemeinde/Ortsteil) (Stadtteil/Gemeindeteil)

### Angaben zum Gebäude

**Gebäudeart** (501):
- ☐ Einfamilienhaus (101)
- ☐ Siedlungshaus (102)
- ☐ Villa, Landhaus (103)
- ☐ Reihenhaus (104)
- ☐ Haus einer Hausgruppe (105) Gartenhof-, Atrium-, Kettenhaus
- ☐ Zweifamilienhaus (106)
- ☐ Wochenendhaus (108)
- ☐ Bauernhaus (109)
- ☐ _____ (sonstige Gebäudeart)

**Anzahl der Wohnungen im Gebäude** (522): ⌊⌋

**Stellung des Gebäudes** (503):
- ☐ freistehendes Haus (1)
- ☐ Doppelhaushälfte (2)
- ☐ Mittelhaus (3)
- ☐ Endhaus (4)

**Ist das Gebäude ein Abbruchobjekt** (304)?
- ☐ nein ☐ ja

**Bei „ja" brauchen die weiteren Fragen nicht beantwortet werden!**

**Baujahr** (504): falls nicht bekannt, ca. Angabe ⌊⌋⌊⌋⌊⌋⌊⌋

**Ist das Gebäude in den Jahren vor dem Erwerb durchgreifend umgebaut, erneuert oder erweitert worden?**
- ☐ nein ☐ ja, im Jahre (505): ⌊⌋⌊⌋⌊⌋⌊⌋

**Art der Veränderung** (572):
- ☐ Anbau ☐ Anschlusserneuerung ☐ Aufstockung
- ☐ Ausbau ☐ Dachgauben ☐ Leitungserneuerung
- ☐ Sanitäranlagen, neue Installationen ☐ Unterfangung
- ☐ Strukturveränderungen ☐ Vormauerschale
- ☐ Wärmedämmputz ☐ Wärmedämmung ☐ Wintergarten

sonstige Art der Veränderung _____

**Gebäudekonstruktion** (507):
- ☐ Holzgebäude (1)
- ☐ Fachwerkgebäude (2)
- ☐ Gebäude leichter Bauart (3)
- ☐ Fertighaus in leichter Bauart (4)
- ☐ Fertighaus in massiver Bauart (5)
- ☐ Mauerwerksbau (6)
- ☐ Nurdachhaus (7)
- ☐ Stahlbeton-, Stahl- oder Stahlbetonskelettbau (8)
- ☐ Holzskelettbau (9)
- ☐ _____ (sonst. Konstruktion)

**Aufbau der Außenmauern** (508) **und Fassade** (509):

Außenmauer:
- ☐ einschalig ohne Wärmedämmung (1)
- ☐ einschalig mit Wärmedämmung) (2)
- ☐ zweischalig (3)

Fassade:
- ☐ einfacher glatter Putz, Plattenwände (1)
- ☐ einfacher Putz, Kalksandstein gefugt (3)
- ☐ Edelputz, Sockel in Klinker, Riemchen (5)
- ☐ Klinker, Keramikplatten, Glasverkleidung (7)
- ☐ Naturstein, Spaltklinker; Mosaik (9)

_____ (sonst. Fassade)

**Dachform** (510):
- ☐ Flachdach (1)
- ☐ Pultdach (2)
- ☐ Sattel-, Krüppelwalmdach (3)
- ☐ Walmdach (4)
- ☐ Mansardendach (5)
- ☐ Zelt-, Kegel- oder Kuppeldach (6)
- ☐ Bogen-, Tonnendach (7)
- ☐ sonstige Dachform (9)

_____

**Zahl der Vollgeschosse (siehe Skizze)** (511)
**ohne Keller und ohne Dachgeschoss (DG)**: ⌊⌋

**Keller** (512):
- ☐ nein (0) ☐ vollständig (100) ☐ tlw., ca. _____ %

davon zum Wohnen ausgebaut, ca. _____ %

**Ausbau des Dachgeschosses** (513):
- ☐ nein (0) ☐ vollständig (100) ☐ tlw., ca. _____ %

**Dachausbau möglich?** ☐ nein ☐ ja

**Drempelhöhe (Kniestock) im Dachgeschoss**:
- ☐ nicht vorhanden ☐ vorhanden, Höhe: ⌊⌋,⌊⌋⌊⌋ m

**Wohn- und Gewerbeflächen im Gebäude** (516, 517):

Wohnfläche (516) ⌊⌋⌊⌋⌊⌋⌊⌋ m²

Gewerbefläche (517) ⌊⌋⌊⌋⌊⌋⌊⌋ m² _____
(nicht das häusliche Arbeitszimmer) (Art der gewerbl. Nutzung z.B. Praxis, Büro...)

**Anzahl der Einstellplätze**:

bei Garage im Gebäude (524) ⌊⌋⌊⌋
bei Garagen als Nebengebäude (525) ⌊⌋⌊⌋
bei Carports (574) ⌊⌋⌊⌋
bei Stellplätzen (526) ⌊⌋⌊⌋

**Sind außer Garagen und Carports weitere Gebäude auf dem Grundstück vorhanden und von Einfluss auf den Kaufpreis?** (502):
- ☐ nein ☐ ja, und zwar _____

geschätzter Wert der Gebäude: ⌊⌋⌊⌋⌊⌋⌊⌋⌊⌋ Euro

### Anlage 2: gRPC-API Teil I

```
syntax = "proto3";
package grpcform;
option go_package = "grpcform";

service FormService {
    rpc GetForm(GetFormRequest) returns (Form) {}
    rpc ValidateForm(Form) returns (Form) {}
    rpc SendForm(Form) returns (SendFormResponse) {}
}

message GetFormRequest {
    string name = 1;
}

message SendFormResponse {
    Form form = 1;
    bool succeed = 2;
    string message = 3;
}

message Form {
    string name = 1;
    repeated Field fields = 2;
    repeated Button buttons = 3;
    bool valid = 4;
}

message Field {
    string label = 1;
    FieldStatus status = 2;
    bool instant_validate = 3;
    string error = 4;
    TextField text_field = 10;
    SelectField select_field = 11;
    NumericField numeric_field = 12;
    ActiveIf activeIf = 20;
    RequiredIf required_if = 21;
    DisabledIf disabled_if = 22;
    HiddenIf hidden_if = 23;
}

enum FieldStatus {
    FIELD_STATUS_UNSPECIFIED = 0;
    FIELD_STATUS_ACTIVE = 1;
    FIELD_STATUS_REQUIRED = 2;
    FIELD_STATUS_DISABLED = 3;
    FIELD_STATUS_HIDDEN = 4;
}


message ActiveIf {
    repeated Validator validators = 1;
}
message RequiredIf {
    repeated Validator validators = 1;
}
message DisabledIf {
     repeated Validator validators = 1;
}
message HiddenIf {
    repeated Validator validators = 1;
}
```

## Anlage 3: gRPC-API Teil II

```
message Validator {
    int64 index = 1;
    string text_is_equal = 10;
    int64 length_smaller_than = 11;
    int64 length_greater_than = 12;
    int64 number_is_equal = 20;
    int64 number_smaller_than = 21;
    int64 number_greater_than = 22;
    string match_regex_pattern = 30;
}

message TextField {
    string value = 1;
    repeated Option options = 2;
    int64 min = 10;
    string min_error = 20;
    int64 max = 11;
    string max_error = 21;
    string regex = 12;
    string regex_error = 22;
}

message SelectField {
    int64 index = 1;
    SelectType type = 2;
    repeated Option options = 3;
    string error = 4;
}

message Option {
    int64 index = 1;
    string value = 2;
}

enum SelectType {
    SELECT_TYPE_UNSPECIFIED = 0;
    SELECT_TYPE_SIMPLE = 1;
    SELECT_TYPE_MULTI = 2;
}

message NumericField {
    int64 value = 1;
    int64 step = 2;
    int64 min = 10;
    string min_error = 20;
    int64 max = 11;
    string max_error = 21;
}
```

## Anlage 4: gRPC-API Teil III

```
message Button {
    string label = 1;
    ButtonStatus status = 2;
    ButtonFuncType type = 3;
}

enum ButtonStatus {
    BUTTON_STATUS_UNSPECIFIED = 0;
    BUTTON_ACTIVE = 1;
    BUTTON_DISABLED = 3;
    BUTTON_HIDDEN = 4;
}

enum ButtonFuncType {
    BUTTON_FUNC_UNSPECIFIED = 0;
    BUTTON_FUNC_VALIDATE = 1;
    BUTTON_FUNC_RESET = 2;
    BUTTON_FUNC_SEND = 3;
}
```

### Anlage 5: Angular Library Component Teil I

```
@Component({
   selector: 'ng-mat-grpc-form',
   template: `…`,
   styles: []
})
export class NgMatGrpcFormComponent implements OnInit {
   FIELD_STATUS = FieldStatus;
   BUTTON_STATUS = ButtonStatus;
   SELECT_TYPE = SelectType;
   @Input() form: Form;
   @Input() name: string;
   @Output() success = new EventEmitter<string>();

   client: FormServiceClient;

   constructor() {
      this.form = new Form();
   }

   ngOnInit() {
      this.client = new FormServiceClient(this.host);
      this.get();
   }

   do(buttonFunc: number): void {
      if (buttonFunc == ButtonFuncType.BUTTON_FUNC_SEND) {
         this.send();
         return
      }
      if (buttonFunc == ButtonFuncType.BUTTON_FUNC_RESET) {
         this.get();
         return
      }
      if (buttonFunc == ButtonFuncType.BUTTON_FUNC_VALIDATE) {
         this.validate();
         return
      }
   }

   get(): void {
      const req = new GetFormRequest();
      req.setName(this.name);
      this.client.getForm(req, null, (err, form) => {
         if (err) {
            console.log(err);
            return;
         }
         this.form = form;
      });
   }
   ...
```

## Anlage 6: Angular Library Component Teil II

```
    validateField(field: Field): void {
        if (field.getInstantValidate()) {
            this.validate();
        }
        let fields = this.form.getFieldsList();
        fields.forEach(function(field) {
            if (field.getInstantValidate()) {
                return
            }
            let textField = field.getTextField();
            if (textField) {
                if (textField.getValue().length < textField.getMin()) {
                    field.setError(textField.getMinError());
                    return
                }
                if (textField.getValue().length > textField.getMax()) {
                    field.setError(textField.getMaxError());
                    return
                }
            }
            field.setError("");
        });
    }

    validate() {
        if (this.host == undefined) {
            let self = this;
            this.form.getFieldsList().forEach(function(field) {
                self.validateField(field)
            });
            return
        }
        this.client.validateForm(this.form, null, (err, form) => {
            if (err) {
                console.log(err);
                return;
            }
            this.form = form;
        });
    }

    send(): void {
        if (this.host == undefined) {
            this.success.emit("Thanks")
            return
        }
        this.client.sendForm(this.form, null, (err, res) => {
            if (err) {
                console.log(err);
                return;
            }
            this.form = res.getForm();
            if (res.getSucceed()) {
                this.success.emit(res.getMessage());
            }
        });
    }
}
```

### Anlage 7: Angular Library Component Template Teil I

```
<div class="ng-mat-grpc-form mat-typography">
    <div *ngFor="let field of form.getFieldsList(); index as i"
         [hidden]="field.getStatus() == FIELD_STATUS.FIELD_STATUS_HIDDEN">
        <div *ngIf="field.getTextField(); let textField">
            <mat-form-field (change)="textField.setValue($event.target.value);
                validateField(field)">
                <input matInput [placeholder]="field.getLabel()"
                       [value]="textField.getValue()" [matAutocomplete]="auto"
                       [attr.disabled]="field.getStatus() ==
                            FIELD_STATUS.FIELD_STATUS_DISABLED"
                       [required]="field.getStatus() ==
                            FIELD_STATUS.FIELD_STATUS_REQUIRED">
                <mat-autocomplete
                    (optionSelected)="textField.setValue($event.option.value);
                        validateField(field)"
                    #auto="matAutocomplete">
                    <mat-option *ngFor="let option of textField.getOptionsList()"
                                [value]="option.getValue()">
                        {{ option.getValue() }}
                    </mat-option>
                </mat-autocomplete>
            </mat-form-field>
        </div>
        <div *ngIf="field.getSelectField(); let select">
            <div *ngIf="select.getType() == SELECT_TYPE.SELECT_TYPE_MULTI">
                <mat-form-field>
                    <mat-label>{{ field.getLabel() }}</mat-label>
                    <mat-select (selectionChange)="select.setIndex($event.value);
                            validateField(field)" [value]="select.getIndex()"
                        [disabled]="field.getStatus() ==
                            FIELD_STATUS.FIELD_STATUS_DISABLED"
                        [required]="field.getStatus() ==
                            FIELD_STATUS.FIELD_STATUS_REQUIRED">
                        <mat-option *ngFor="let option of select.getOptionsList()"
                                    [value]="option.getIndex()">
                            {{ option.getValue() }}
                        </mat-option>
                    </mat-select>
                </mat-form-field>
            </div>
            <div *ngIf="select.getType() == SELECT_TYPE.SELECT_TYPE_SIMPLE">
                <div *ngIf="field.getLabel()">
                    <label>{{ field.getLabel() }}<br></label>
                </div>
                <mat-radio-group (change)="select.setIndex($event.value);
                        validateField(field)"
                    [value]="select.getIndex()" [disabled]="field.getStatus() ==
                        FIELD_STATUS.FIELD_STATUS_DISABLED"
                    [required]="field.getStatus() ==
                        FIELD_STATUS.FIELD_STATUS_REQUIRED">
                    <mat-radio-button *ngFor="let option of select.getOptionsList()"
                                      [value]="option.getIndex()">
                        {{ option.getValue() }}
                    </mat-radio-button>
                </mat-radio-group>
            </div>
        </div>
    ...
```

## Anlage 8: Angular Library Components Template Teil II

```
        <div *ngIf="field.getNumericField(); let numericField">
            <div *ngIf="field.getLabel()">
                <label>{{ field.getLabel() }}<br></label>
            </div>
            <mat-slider (change)="numericField.setValue($event.value);
                    validateField(field)" thumbLabel
               [step]="numericField.getStep()" [min]="numericField.getMin()"
               [max]="numericField.getMax()" [value]="numericField.getValue()"
               [disabled]="field.getStatus() ==FIELD_STATUS.FIELD_STATUS_DISABLED">
            </mat-slider>
        </div>
        <mat-error>
            {{ field.getError() }}
        </mat-error>
    </div>
    <div *ngFor="let button of form.getButtonsList()">
        <button *ngIf="button.getStatus() != BUTTON_STATUS.BUTTON_HIDDEN"
            (click)="do(button.getType())" [disabled]="button.getStatus() ==
                BUTTON_STATUS.BUTTON_DISABLED"
                mat-button>{{ button.getLabel() }}</button>
    </div>
</div>
```

## Anlage 9: Angular Library Module

```
import { NgModule } from '@angular/core';
import { NgMatGrpcFormComponent } from './ng-mat-grpc-form.component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { CommonModule} from "@angular/common";
import { BrowserAnimationsModule } from "@angular/platform-browser/animations";
import { HttpClientModule } from "@angular/common/http";
import {
  MatAutocompleteModule, MatButtonModule, MatFormFieldModule, MatInputModule,
  MatOptionModule, MatRadioModule, MatSelectModule, MatSliderModule
} from "@angular/material";

@NgModule({
  declarations: [NgMatGrpcFormComponent],
  imports: [
    CommonModule,
    BrowserAnimationsModule,
    FormsModule,
    RouterModule,
    ReactiveFormsModule,
    HttpClientModule,
    MatInputModule,
    MatFormFieldModule,
    MatButtonModule,
    MatRadioModule,
    MatSelectModule,
    MatAutocompleteModule,
    MatSliderModule,
    MatOptionModule
  ],
  exports: [NgMatGrpcFormComponent]
})
export class NgMatGrpcFormModule { }
```

### Anlage 10: Go-gRPC-API Teil I

```go
package grpcform

import (
   "context"
   "net"
   "regexp"
   "strconv"
   "sync"

   "google.golang.org/grpc"
   "google.golang.org/grpc/reflection"
)

type ModelFunc func() *Form
type SendFunc func(context.Context, *Form) (*SendFormResponse, error)

func New() ProxyServer {
   return make(map[string]element)
}

func (s ProxyServer) Add(model ModelFunc, send SendFunc) {
   safe.Lock()
   s[model().GetName()] = element{model: model, send: send}
   safe.Unlock()
}

var safe sync.Mutex

type ProxyServer map[string]element

type element struct {
   model ModelFunc
   send  SendFunc
}

func (s ProxyServer) Start(host string) error {
   lis, err := net.Listen("tcp", host)
   if err != nil {
      return err
   }
   gs := grpc.NewServer()
   RegisterFormServiceServer(gs, s)
   reflection.Register(gs)
   if err := gs.Serve(lis); err != nil {
      return err
   }
   return nil
}

func (s ProxyServer) GetForm(ctx context.Context, req *GetFormRequest) (
   *Form,
   error,
) {
   safe.Lock()
   defer safe.Unlock()
   for _, e := range s {
      if req.GetName() == e.model().GetName() {
         return e.model(), nil
      }
   }
   return &Form{}, nil
}
```

## Anlage 11: Go-gRPC-API Teil II

```
func (s ProxyServer) ValidateForm(ctx context.Context, in *Form) (
   *Form,
   error,
) {
   out, err := s.GetForm(ctx, &GetFormRequest{Name: in.GetName()})
   if err != nil || in == nil || len(out.GetFields()) != len(in.GetFields()) {
      return &Form{}, nil
   }
   out.Valid = true
   outFields := out.GetFields()
   inFields := in.GetFields()
   for i, inField := range inFields {
      outField := outFields[i]
      if activeIf := outField.GetActiveIf(); activeIf != nil {
         checkValidator(inFields, outField, activeIf.GetValidators(),
            FieldStatus_FIELD_STATUS_ACTIVE)
      }
      if requiredIf := outField.GetRequiredIf(); requiredIf != nil {
         checkValidator(inFields, outField, requiredIf.GetValidators(),
            FieldStatus_FIELD_STATUS_REQUIRED)
      }
      if disabledIf := outField.GetDisabledIf(); disabledIf != nil {
         checkValidator(inFields, outField, disabledIf.GetValidators(),
            FieldStatus_FIELD_STATUS_DISABLED)
      }
      if hiddenIf := outField.GetHiddenIf(); hiddenIf != nil {
         checkValidator(inFields, outField, hiddenIf.GetValidators(),
            FieldStatus_FIELD_STATUS_HIDDEN)
      }
      if inTextField := inField.GetTextField(); hasStatus(outField.GetStatus(),
         FieldStatus_FIELD_STATUS_ACTIVE, FieldStatus_FIELD_STATUS_REQUIRED) &&
         inTextField != nil {
         outTextField := outField.GetTextField()
         outTextField.Value = inTextField.GetValue()
         if !out.Valid ||
            outField.GetStatus() == FieldStatus_FIELD_STATUS_UNSPECIFIED ||
            (outField.GetStatus() == FieldStatus_FIELD_STATUS_ACTIVE &&
               outTextField.Value == "") {
            continue
         }
         if int64(len(outTextField.GetValue())) < outTextField.GetMin() {
            outField.Error = outTextField.GetMinError()
            out.Valid = false
            continue
         }
         if int64(len(outTextField.GetValue())) > outTextField.GetMax() {
            outField.Error = outTextField.GetMaxError()
            out.Valid = false
            continue
         }
         if ok, err := regexp.MatchString(outTextField.GetRegex(),
            outTextField.GetValue()); !ok || err != nil {
            outField.Error = outTextField.GetRegexError()
            out.Valid = false
            continue
         }
      }
  ...
```

## Anlage 12: Go-gRPC-API Teil III

```
if inSelectField := inField.GetSelectField(); hasStatus(outField.GetStatus(),
        FieldStatus_FIELD_STATUS_ACTIVE, FieldStatus_FIELD_STATUS_REQUIRED) &&
        inSelectField != nil {
        outSelectField := outField.GetSelectField()
        outSelectField.Index = inSelectField.GetIndex()
        if !out.Valid || (outField.GetStatus() == FieldStatus_FIELD_STATUS_ACTIVE
&&
           outSelectField.GetIndex() == 0) {
           continue
        }
        check := false
        for _, o := range outSelectField.GetOptions() {
           if o.GetIndex() == outSelectField.GetIndex() {
              check = true
              continue
           }
        }
        if !check {
           outField.Error = outSelectField.GetError()
           out.Valid = false
           continue
        }
     }
     if inNumericField := inField.GetNumericField();
hasStatus(outField.GetStatus(),
        FieldStatus_FIELD_STATUS_ACTIVE, FieldStatus_FIELD_STATUS_REQUIRED) &&
        inNumericField != nil {
        outSlider := outField.GetNumericField()
        outSlider.Value = inNumericField.GetValue()
        if !out.Valid || (outField.GetStatus() == FieldStatus_FIELD_STATUS_ACTIVE
&&
           outSlider.Value == 0) {
           continue
        }
        v := outSlider.GetValue()
        if int64(v) < outSlider.GetMin() {
           outField.Error = outSlider.GetMinError()
           out.Valid = false
           continue
        }
        if int64(v) > outSlider.GetMax() {
           outField.Error = outSlider.GetMaxError()
           out.Valid = false
           continue
        }
     }
  }
  if out.GetValid() {
     for _, b := range out.GetButtons() {
        b.Status = ButtonStatus_BUTTON_ACTIVE
     }
  }
  return out, nil
}
```

## Anlage 13: Go-gRPC-API Teil IV

```
func (s ProxyServer) SendForm(ctx context.Context, in *Form) (
   res *SendFormResponse,
   err error,
) {
   out, err := s.ValidateForm(ctx, in)
   if err != nil {
      return &SendFormResponse{Form: out}, nil
   }
   return s[out.GetName()].send(ctx, out)
}

func checkValidator(inFields []*Field, outField *Field, validators []*Validator,
status FieldStatus) {
   for _, validator := range validators {
      index := validator.GetIndex()
      if textField := inFields[index].GetTextField(); textField != nil &&
         checkValidatorOnTextField(textField, validator) {
         outField.Status = status
         break
      }

      if numericField := inFields[index].GetNumericField(); numericField != nil &&
         checkValidatorOnNumericField(numericField, validator) {
         outField.Status = status
         break
      }
      if selectField := inFields[index].GetSelectField(); selectField != nil &&
         checkValidatorOnSelectField(selectField, validator) {
         outField.Status = status
         break
      }
   }
}

func checkValidatorOnTextField(
   textField *TextField,
   validator *Validator,
   ) bool {
   if v := validator.GetTextIsEqual(); v != "" &&
      textField.GetValue() == v {
      return true
   }
   if v := validator.GetLengthSmallerThan(); v != 0 &&
      int64(len(textField.GetValue())) < v {
      return true
   }
   if v := validator.GetLengthGreaterThan(); v != 0 &&
      int64(len(textField.GetValue())) > v {
      return true
   }
   if v := validator.GetMatchRegexPattern(); v != "" {
      if ok, err := regexp.MatchString(v, textField.GetValue()); ok &&
         err != nil {
         return true
      }
   }
   return false
}
```

## Anlage 14: Go-gRPC-API Teil V

```
func checkValidatorOnNumericField(numericField *NumericField, validator *Validator)
bool {
   if v := validator.GetNumberIsEqual(); v != 0 && numericField.GetValue() == v {
      return true
   }
   if v := validator.GetNumberSmallerThan(); v != 0 && numericField.GetValue() < v
{
      return true
   }
   if v := validator.GetNumberGreaterThan(); v != 0 && numericField.GetValue() > v
{
      return true
   }
   if v := validator.GetMatchRegexPattern(); v != "" {
      if ok, err := regexp.MatchString(v,
strconv.Itoa(int(numericField.GetValue()))); ok && err != nil {
         return true
      }
   }
   return false
}

func checkValidatorOnSelectField(selectField *SelectField, validator *Validator)
bool {
   if text := validator.GetTextIsEqual(); text != "" {
      if getOption(selectField.GetIndex(), selectField.GetOptions()) != nil {
         return true
      }
   }
   if v := validator.GetNumberIsEqual(); v != 0 && selectField.GetIndex() == v {
      return true
   }
   if v := validator.GetNumberSmallerThan(); v != 0 && selectField.GetIndex() < v {
      return true
   }
   if v := validator.GetNumberGreaterThan(); v != 0 && selectField.GetIndex() > v {
      return true
   }
   if regex := validator.GetMatchRegexPattern(); regex != "" {
      if ok, err := regexp.MatchString(regex, getOption(selectField.GetIndex(),
         selectField.GetOptions()).GetValue()); ok && err != nil {
         return true
      }
   }
   return false
}

func getOption(option int64, options []*Option) *Option {
   for _, o := range options {
      if o.GetIndex() == option {
         return o
      }
   }
   return nil
}

func hasStatus(is FieldStatus, within ...FieldStatus) bool {
   for _, s := range within {
      if s == is {
         return true
      }
   }
   return false
}
```

## Anlage 15: Go Server Beispiel Teil I

```go
package main

import (
   "context"
   "google.golang.org/grpc"
   "example/backend/api"
   grpcform "github.com/grpc-form/api/go"
)

const (
   USERNAME = iota
   AGE
   CAR
   BRAND
)

var (
   NO         int64 = 1
   YES        int64 = 2
   VOLKSWAGEN int64 = 27
)

func main() {
   conn, err := grpc.Dial("database-service:9000", grpc.WithInsecure())
   if err != nil {
      panic(err)
   }
   db := api.NewDatabaseClient(conn)

   s := grpcform.New()
   s.Add(func() *grpcform.Form {
      return &grpcform.Form{
         Name: "car",
         Fields: []*grpcform.Field{
            USERNAME: {
               InstantValidate: true,
               Label:           "Please enter your username",
               Status:          grpcform.FieldStatus_FIELD_STATUS_REQUIRED,
               TextField: &grpcform.TextField{
                  Min:        5,
                  MinError:   "Too Short",
                  Max:        25,
                  MaxError:   "Too long",
                  Regex:      "^[a-zA-Z]{5,25}$",
                  RegexError: "Only letters",
               },
            },
            AGE: {
               Label:           "How old are you?",
               InstantValidate: true,
               Status:          grpcform.FieldStatus_FIELD_STATUS_REQUIRED,
               NumericField: &grpcform.NumericField{
                  Min:   14,
                  Max:   99,
                  Step:  1,
                  Value: 25,
               },
            },
         ...
```

## Anlage 16: Go Server Beispiel Teil II

```
        CAR: {
            Label:           "Do you have a car?",
            InstantValidate: true,
            Status:          grpcform.FieldStatus_FIELD_STATUS_REQUIRED,
            SelectField: &grpcform.SelectField{
               Index: NO,
               Type:  grpcform.SelectType_SELECT_TYPE_SIMPLE,
               Options: []*grpcform.Option{
                  {Index: NO, Value: "No"},
                  {Index: YES, Value: "Yes"},
               },
            },
            HiddenIf: &grpcform.HiddenIf{
               Validators: []*grpcform.Validator{
                  {
                     Index:             AGE,
                     NumberSmallerThan: 18,
                  },
               },
            },
         },
         BRAND: {
            Status:          grpcform.FieldStatus_FIELD_STATUS_HIDDEN,
            InstantValidate: true,
            Label:           "What kind of car are you driving?",
            SelectField: &grpcform.SelectField{
               Index: VOLKSWAGEN,
               Type:  grpcform.SelectType_SELECT_TYPE_MULTI,
               Options: []*grpcform.Option{
                  {Index: 1, Value: "Audi"},
                  {Index: 2, Value: "BMW"},
                  {Index: 3, Value: "Bentley"},
                  {Index: 4, Value: "Chevrolet"},
                  {Index: 5, Value: "FIAT"},
                  {Index: 6, Value: "Ferrari"},
                  {Index: 7, Value: "Ford"},
                  {Index: 8, Value: "HUMMER"},
                  {Index: 9, Value: "Hyundai"},
                  {Index: 10, Value: "Jaguar"},
                  {Index: 11, Value: "Jeep"},
                  {Index: 12, Value: "Kia"},
                  {Index: 13, Value: "Lamborghini"},
                  {Index: 14, Value: "Maybach"},
                  {Index: 15, Value: "Mazda"},
                  {Index: 16, Value: "McLaren"},
                  {Index: 17, Value: "Mercedes-Benz"},
                  {Index: 18, Value: "Nissan"},
                  {Index: 19, Value: "Opel"},
                  {Index: 20, Value: "Jaguar"},
                  {Index: 21, Value: "Porsche"},
                  {Index: 22, Value: "Renault"},
                  {Index: 23, Value: "Skoda"},
                  {Index: 24, Value: "Suzuki"},
                  {Index: 25, Value: "Tesla"},
                  {Index: 26, Value: "Toyota"},
                  {Index: VOLKSWAGEN, Value: "Volkswagen"},
                  {Index: 28, Value: "Volvo"},
                  {Index: 29, Value: "smart"},
               },
            },
...
```

## Anlage 17: Go Server Beispiel Teil III

```
            RequiredIf: &grpcform.RequiredIf{
               Validators: []*grpcform.Validator{
                  {
                     Index:             AGE,
                     NumberGreaterThan: 17,
                  },
                  {
                     Index:         CAR,
                     NumberIsEqual: YES,
                  },
               },
            },
            HiddenIf: &grpcform.HiddenIf{
               Validators: []*grpcform.Validator{
                  {
                     Index:             AGE,
                     NumberSmallerThan: 18,
                  },
                  {
                     Index:         CAR,
                     NumberIsEqual: NO,
                  },
               },
            },
         },
      },
      Buttons: []*grpcform.Button{
         {
            Label:  "Reset",
            Type:   grpcform.ButtonFuncType_BUTTON_FUNC_RESET,
            Status: grpcform.ButtonStatus_BUTTON_ACTIVE,
         },
         {
            Label:  "Send",
            Type:   grpcform.ButtonFuncType_BUTTON_FUNC_SEND,
            Status: grpcform.ButtonStatus_BUTTON_DISABLED,
         },
      },
      Valid: false,
   }
}, ...
```

## Anlage 18: Go Server Beispiel Teil IV

```
func(ctx context.Context, in *grpcform.Form) (res *grpcform.SendFormResponse, err
error) {
      out, err := s.ValidateForm(ctx, in)
      if out == nil || err != nil || !out.GetValid() {
         return &grpcform.SendFormResponse{
            Form:    out,
            Succeed: false,
            Message: "Insert Failed: Not Valid",
         }, nil
      }
      index := out.GetFields()[BRAND].GetSelectField().GetIndex()
      options := out.GetFields()[BRAND].GetSelectField().GetOptions()
      var car string
      for _, o := range options {
         if o.GetIndex() == index {
            car = o.GetValue()
         }
      }
      u, err := db.InsertUser(ctx, &api.User{
         Name: out.GetFields()[USERNAME].GetTextField().GetValue(),
         Age:  out.GetFields()[AGE].GetNumericField().GetValue(),
         Car:  car,
      })
      if u == nil {
         return &grpcform.SendFormResponse{
            Form:    out,
            Succeed: false,
            Message: "Insert Failed: DB Error",
         }, nil
      }
      if u.GetName() != out.GetFields()[USERNAME].GetTextField().GetValue() {
         out.GetFields()[USERNAME].Error = "You have already completed this form"
         return &grpcform.SendFormResponse{
            Form:    out,
            Succeed: false,
            Message: "Insert Failed: You have already completed this form",
         }, nil
      }
      return &grpcform.SendFormResponse{
         Form:    out,
         Succeed: true,
         Message: "Thank you!",
      }, nil
   })
   err = s.Start(":50051")
   conn.Close()
   if err != nil {
      panic(err)
   }
}
```