Fernuniversität Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme

Bachelorarbeit

Definitionsgetriebene Softwareentwicklung

Lösungsansatz eines definitionsgetriebenen Front- und Backendseitigen Frameworks für rollenbasierte Datenbank-Anwendungen in einer serviceorientierten Architektur

Ismaiel, Samy
Zaubernussweg 20
48531 Nordhorn
samy@ismaiel.de
(0 59 21) 7 27 79 88
Matrikel-Nr. 8939489

Studiengang:	Bachelor Informatik
Semester:	Sommer 2023
Betreuende:	Dr. Daniela Keller

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

„Boilerplate" – diesen Begriff prägte zunächst die Druckindustrie für Zeitungskolumnen, die mit selbem Wortlaut in mehreren Printmedien erscheinen und deren Druckvorlagen an Metallplatten zur Herstellung von Boilern erinnern. In der Softwareentwicklung wird dieser Ausdruck auf Quelltext angewandt, der aus vielen Wiederholungen nahezu identischer Befehlssequenzen besteht [1].

Dass der Output eines kreativen Prozesses wie der Programmierung bisweilen mit diesem Attribut versehen wird, zeigt für sich schon ein Problem und ein entsprechendes Problembewusstsein innerhalb der Software-produzierenden Zunft auf.

Dabei ist derlei Code selbst in Lehrbüchern zu finden. So enthält beispielsweise das im Jahr 2021 erschienene Buch *Beginning Java MVC 1.0* [2] auf knapp 70 Seiten seines zwölften und letzten Kapitels ein Musterbeispiel für das erst vor wenigen Jahren verabschiedete Framework *Java MVC 1.0* [3]. Die in o. g. Lehrbuch enthaltenen rund 1.500 Zeilen Quellcode realisieren eine kleine Bibliotheksverwaltung mit nur drei Entitätstypen für Bücher, Mitglieder und deren Ausleihen sowie grundlegende CRUD-Operationen hierzu[1].

Ein solches Missverhältnis zwischen der Menge an Quellcode und dem Umfang an Funktionen ist der Vielzahl von Code-Zeilen geschuldet, die als „boilerplate" bezeichnet werden können. Pro Entitätstyp werden fünf Klassen benötigt: jeweils eine für Entity, Persistence, Modell, Controller und View.

Für jedes Attribut einer Entity-Klasse wiederum benötigt man eine Instanz-Variable, die mindestens eine, größtenteils jedoch mehrere Annotationen erfordert, sowie eine Getter- und eine Setter-Methode. Ähnlich stereotyp sind die übrigen Klassen gestaltet.

Getter- und Setter-Methoden lassen sich in modernen Entwicklungsumgebungen wie Eclipse auf Knopfdruck generieren. Gängige CASE-Tools[2] ermöglichen es sogar, größere Artefakte, wie etwa ganze Entity-Klassen, zu generieren – etwa an

---

1 CRUD steht für Create, Read, Update und Delete und fasst damit grundlegende Datenoperationen zu einem Begriff zusammen.

2 CASE = Computer Aided Software Engeneering

Hand des zugrundeliegenden Datenmodells oder per Reverse Engineering aus einer konkreten Datenbank [4].

Dies wäre jedoch allenfalls eine symptomatische Behandlung des Problems der vielen Schreibarbeit, die durch ein Vorgehen verursacht wird, an dessen Ende boilerplate Code steht, denn prononciert formuliert gilt: „Generierter Code ist dummer Code."

Ein Generator ist stets intelligenter als sein Ergebnis, da eine Generierung einen klassischen EVA-Prozess[3] darstellt, dessen Verarbeitungsteil die Logik der zu generierenden Ausgabe kennen muss und somit mindestens so intelligent sein muss wie eben diese Ausgabe. Hinzu kommt, dass der Generator neben Lese- und Schreiboperationen die Analyse der Eingabe sowie die Synthese der Ausgabe beherrschen muss – inkl. sämtlicher Fallunterscheidungen, die von der Eingabe zur Ausgabe führen.

Zudem sind große Mengen an generiertem Code ein Problem für sich. Code will verwaltet werden und er unterliegt einem stetigen Änderungsprozess, da bei Änderungen auf der Eingabe-Seite des EVA-Prozesses eine neue Ausgabe erwartet wird.

Und schlussendlich wird man sich auch nicht mit generierten View-Klassen begnügen. Interdependenzen zwischen den Bedienelementen sowie Logik, die über grundlegende CRUD-Operationen hinausgeht, muss so ergänzt werden können, dass ein erneuter Generierungslauf diese nicht überschreibt.

Ein effizienter Entwicklungsprozess ist somit kaum dadurch zu realisieren, dass man boilerplate Code generieren lässt, sondern dadurch, dass man ihn entbehrlich macht.

Eine Herangehensweise, wie man dies für Web-basierte Datenbank-Anwendungen realisieren kann, möchte ich in dieser Bachelorarbeit vorstellen.

[3] EVA = Eingabe Verarbeitung Ausgabe

1.2 Methodik

Hierbei wird ein ganzheitlicher Ansatz verfolgt, der sowohl die Frontend- als auch die Backend-Seite einer serviceorientierten Architektur betrachtet. Kernstück des beschriebenen Ansatzes sind Definitionen, mit denen die Entwicklerin angibt, was die Anwendung verarbeiten und darstellen soll, wobei das „Wie" einem Framework überlassen wird, das diese Definitionen zur Laufzeit interpretiert und sowohl die GUI daraus ableitet als auch die DML-Statements[4] für CRUD-Operationen unmittelbar vor deren Ausführung generiert und absetzt, ohne dass diese Teil des zu verwaltenden Quellcode-Umfangs werden.

Neben der Tatsache, dass die Entwicklerin von der Implementierung typischer CRUD-Operationen entlastet wird, soll gezeigt werden, dass diese Vorgehensweise auch die Umsetzung eines Rollenkonzepts vereinfacht.

Sowohl frontend- als auch backendseitig sind bei der Implementierung eines solchen Frameworks mehrere Szenarien abzudecken, für die geeignete Definitionen vorgestellt werden.

Hierbei erfolgt die Beschreibung vom Allgemeinen zum Speziellen – also von der Definition grundlegender Funktionalität hin zu vertiefenden Überlegungen.

Die beschriebenen Szenarien werden an Hand von Beispielen verdeutlicht. Zur besseren Vergleichbarkeit der Entwicklung per Codierung vs. Definition wird auf die im Abschnitt *Motivation* beschriebene Anwendung in [2] Bezug genommen, indem zunächst eine funktional identische Anwendung definiert wird. Dafür werden rund 330 Zeilen Definition an Stelle der o. g. 1.500 Zeilen Quellcode benötigt.

Im Anschluss wird die Anwendung ausgebaut, um weitere erforderliche Definitionsmöglichkeiten des hier beschriebenen Frameworks zu verdeutlichen.

---

[4] DML = Data Manipulation Language; DML umfasst den Teil des Sprachumfangs von SQL, der mit der Bearbeitung von Daten zu tun hat (also die Befehle INSERT, SELECT, UPDATE und DELETE). Im Unterschied hierzu umfasst SQL noch DDL (Data Definition Language), womit Datenbankobjekte (Tabellen, Views etc.) erzeugt und geändert werden (also die Befehle CREATE, ALTER und DROP), sowie DCL (Data Control Language), womit Berechtigungen vergeben und entzogen werden können (also die Befehle GRANT und REVOKE). Vorgenannte Aufzählung von Befehlen erhebt keinen Anspruch auf Vollständigkeit.

1.3 Themen-Abgrenzung

Ein wie oben beschrieben auf Definitionen basierendes Framework habe ich Anfang 2019 für die Entwicklungsumgebung Gupta TD Mobile entwickelt und Ende 2019 auf der Gupta Entwicklerkonferenz [5] inkl. der darauf basierend erstellten Fuhrparkmanagement-Software Fleet+ Web vorgestellt.

Ende 2020 habe ich das Backend dieses Frameworks in Python reimplementiert, wobei die Definitionen, die die eigentliche Anwendung darstellen, beibehalten werden konnten. Auch das von mir vollständig in JavaScript geschriebene Frontend des Frameworks konnte beibehalten werden. Inzwischen entwickelt bei meinem Arbeitgeber Carano Software Solutions GmbH ein 9-köpfiges Team mit diesem von mir erstellten und durch mich gepflegten Framework an Fleet+ Web.

Die bereits vorliegenden Implementierungen eines solchen Frameworks sollen nicht Gegenstand dieser Arbeit sein. Vielmehr soll es darum gehen, die benötigten Definitionsmöglichkeiten zu beschreiben. Dabei wird die bei Fleet+ Web verwendete Syntax für Definitionen genutzt. Im Vordergrund steht jedoch nicht diese Syntax, sondern die Beschreibung der abzudeckenden Anforderungen inkl. der Erklärung, wie sich diese durch Definitionen umsetzen lassen.

Alternative syntaktische Möglichkeiten werden in einem Exkurs im Anhang angesprochen.

Neben einem Rollenkonzept lassen sich in einem solchen Framework auch andere nichtfunktionale Anforderungen wie Datenbankherstellerunabhängigkeit, Konfigurierbarkeit, Internationalisierung, Protokollierung und Datensatzsperren realisieren, was auch mit o. g. Framework umgesetzt wurde, jedoch nicht Gegenstand des Themenschwerpunktes dieser Arbeit ist, während das Rollenkonzept integraler Bestandteil der vorgestellten Herangehensweise ist, da es hier um den definitionsbasierten Ansatz geht und die Definitionen festlegen, wer, wie auf welche Daten zugreifen darf.

Als architektonische Grundlage bieten sich für die Kommunikation zwischen Client und Server REST-Services an. Die detaillierte Implementierung derselben ist nicht Bestandteil dieser Arbeit, da die hier dargestellten Konzepte prinzipiell auch anders realisierbar sind. Es wird auch keine Empfehlung der verwendeten Tools ausgesprochen. Die oben erwähnte Implementierung des Backends mit Python nutzt das Framework FastAPI, mit dem solche Endpunkte leicht entwickelt werden können. Es bietet u. a. die Möglichkeit, dass Endpunkte einer Authentisierung

bedürfen, die über einen JWT-Token[5] im Anfragekopf erfolgt und per Dependency Injection[6] an den Endpunkt weitergereicht wird [6].

In diesem Dokument wird davon ausgegangen, dass für das Frontend JavaScript verwendet wird.

---

5 Ein JWT oder JSON Web Token ist eine verschlüsselte Zeichenkette mit Authentifizierungsdaten.

6 Unter Dependency Injection versteht man die Bereitstellung von Abhängigkeiten bei der Initialisierung von Objekten. Benötigt beispielsweise die beim Aufruf eines Endpunkts aufgerufene Operation ein Objekt mit den Anmeldeinformationen zum aufrufenden Anwender, kann dieses von einem Web Framework wie FastAPI erzeugt und der Operation bereitgestellt werden. Hierzu kann man zentral konfigurieren, welche Endpunkt-Operationen ein solches Objekt benötigen und wie dieses erzeugt werden soll, so dass sich nicht jeder Endpunkt selbst darum kümmern muss.

1.4 Ziel dieser Arbeit

Ziel dieser Arbeit ist die Beschreibung von Anforderungen an ein Framework, das es ermöglicht, eine Datenbankanwendung zu definieren statt zu programmieren. Der Fokus liegt dabei auf den hierfür benötigten Typen von Definitionen und deren Inhalten.

Zur Demonstration des Zusammenspiels eines solchen Frameworks und entsprechender Definitionen wird im Verlauf dieser Arbeit sukzessive eine auf Definitionen basierende Anwendung erstellt – sprich: es werden konkrete Definitionen und das Resultat ihrer Auswertung durch ein solches Framework vorgestellt.

Inhaltlich handelt es sich bei der entstehenden Anwendung wie bei der in [2] beispielhaft implementierten Anwendung um eine Bibliotheksverwaltung. Diese inhaltliche Festlegung hat jedoch nichts mit den vorgestellten Konzepten zu tun, sondern dient lediglich dem Aufbau eines in sich geschlossenen Anwendungsbeispiels. Die Orientierung an der Beispielanwendung in [2] dient zudem der Vergleichbarkeit des Aufwands einer nach den hier vorgestellten Konzepten definierten Anwendung gegenüber einer funktionsgleichen programmierten Applikation.

Auf Basis der hier vorgestellten Anforderungen sollte es möglich sein, ein Low-Code-Framework zu entwickeln, mit dem sich CRUD-Bestandteile einer Datenbankanwendungen ohne Programmierung erstellen lassen. Ein digitaler Karteikasten, wie z. B. die Bibliotheksverwaltung aus [2], ließe sich mittels eines solchen Frameworks ausschließlich durch Definition statt durch Programmierung erstellen. Um neben reiner Karteikastenfunktionalität auch Anwendungslogik umsetzen zu können, bedarf es in einem solchen Framework einiger ebenfalls in dieser Arbeit vorgestellter Erweiterungspunkte, weshalb hier nicht von einem No-Code-Framework, sondern von einem Low-Code-Framework die Rede ist.

1.5 Gender-Hinweis

In diesem Dokument kommen zwei Ausprägungen namentlich nicht bekannter natürlicher Personen vor: Personen, die das hier beschriebene Framework für die Entwicklung benutzen, und solche, die die dadurch entwickelten Anwendungen benutzen oder darin verwaltet werden.

Für erstere wird aus Gründen der sprachlichen Vereinfachung ausschließlich das Femininum verwendet – also Entwicklerin. Für letztere wird durchgängig das Maskulinum verwendet – also Anwender oder Benutzer, wobei mit „Anwender“ der Nutzer der Software (meistens o. g. Beispiel-Bibliotheksverwaltung) gemeint ist und mit „Benutzer“ der in der Bibliotheksverwaltungssoftware verwaltete Nutzer der Bibliothek.

2 Definitionsgetriebener Ansatz

2.1 Einordnung in bekannte Programmierparadigmen

Wie in der Einleitung angedeutet, unterstützt ein nach den Vorschlägen dieses Dokuments entwickeltes Framework eine deklarative Vorgehensweise, da sich die Entwicklerin weniger um das „Wie“, sondern vornehmlich um das „Was“ kümmern muss.

Allerdings lässt sich der hier beschriebene, vom Autor dieser Arbeit „definitionsgetrieben“ genannte Ansatz nicht in einen der drei Teilgebiete der deklarativen Programmierung einordnen [7]. Es geht hier weder um die funktionale Programmierung, bei der „ein Programm als Menge von Funktionen beschrieben [wird], die eine Eingabe eineindeutig auf eine Ausgabe abbildet“, noch um die logische Programmierung oder deren als Constraint Programmierung bezeichnete Weiterentwicklung, bei der Zusicherungen in Form von prädikatenlogischen Formeln angegeben werden. In diesem Papier soll es nicht darum gehen, Zusicherungen zu prüfen, sondern darum, Programmbestandteile in Definitionen auszulagern, die von einem Framework ausgelesen und abgearbeitet werden können.

Auch das als „definitive programming“ bekannte Programmierparadigma, bei dem „Variablen an Stelle von Werten Formeln zugewiesen werden“ [8], entspricht nicht dem hier beschriebenen Vorgehen.

Am ehesten passt der Begriff der „datengetriebenen Programmierung“ zu dem hier vorgestellten Konzept, da „die Programmlogik nicht durch den Programmcode, sondern durch Daten“ [9] gesteuert wird, in diesem Fall Definitionsdateien.

Davon abzugrenzen ist die bisweilen ebenfalls als „datengetrieben“ bezeichnete Abarbeitung eines Datenstroms [10], wie sie mit der Programmiersprache AWK möglich ist, wo in Abhängigkeit von der jeweils eingelesenen Zeile eines Eingabestroms eine Aktion ausgelöst werden kann [11].

Wegen der uneinheitlichen Verwendung des Begriffs „datengetrieben“ wird hier der Begriff „definitionsgetrieben“ bevorzugt. Dieser Begriff hebt hervor, dass es sich bei den steuernden Daten um Definitionen handelt, die das Programmverhalten parametrieren.

2.2 Vorteile des definitionsgetriebenen Ansatzes

Ein definitionsgetriebenes Vorgehen ist von handelsüblichen Reportgeneratoren bekannt, bei denen in einer Reportdefinitionsdatei festgelegt wird, welche Daten ein Bericht ausgeben soll und wie das Layout des Berichts gestaltet sein soll [12].

Eine Anwendung, die mehrere Berichte anbietet, kann auch bei wechselnden Anforderungen an das Berichtswesen relativ statisch bleiben, da lediglich die Reportdefinitionsdateien ausgetauscht werden müssen, um einen Bericht zu ändern, bzw. neue Reportdefinitionsdateien hinzugefügt werden können, um weitere Berichte anzubieten.

Auf eine vergleichbare Weise können in einer Anwendung, die auf einem nach den hier vorgestellten Prinzipien erstellten Framework basiert, auch ohne Programmierung durch bloße Bereitstellung von Definitionsdateien Programmbestandteile erweitert werden.

Programmbestandteile als separat auslieferbare Einheiten (engl. deliverables) bieten neben dem Vorteil von ad hoc realisierbaren Programmänderungen auch den Vorteil, dass das eigentliche Programmgerüst relativ statisch und von solchen Programmänderungen unberührt bleibt, was wiederum zur Robustheit des Gesamtsystems beiträgt, da einerseits das Programmgerüst weniger Veränderungen und damit weniger der Fehlerhaftigkeit unterliegt und andererseits die einzelnen deliverables einen sehr beschränkten Einfluss auf die Stabilität des Gesamtsystems haben. Dieser Effekt leuchtet bei Reports unmittelbar ein, da sich eine fehlerhafte Reportdefinitionsdatei maximal auf den einen betroffenen Berichtstyp auswirkt, aber die Korrektheit der übrigen Berichte oder gar die Lauffähigkeit des Gesamtsystems nicht beeinträchtigt sind.

Für definitionsgetriebene Programmbestandteile gilt die geringere Gefahr von Seiteneffekten bei Änderungen von deliverables grundsätzlich ebenfalls. Es muss jedoch einschränkend auf die potenzielle Möglichkeit von Abhängigkeiten zwischen den Definitionen hingewiesen werden. Änderungen an Definitionen, von denen keine weiteren Definitionen abhängen, erfordern einen geringeren Aufwand für Integrationstests.

Die Vorteile der deklarativen Programmierung, die in der Ausdrucksstärke, im geringeren Code-Umfang und der höheren Zuverlässigkeit liegen [13], können auch mit einem definitionsgetriebenen Framework erzielt werden.

Schwerpunkt dieser Arbeit ist jedoch nicht die Gegenüberstellung von Programmierparadigmen, sondern die Vorstellung einer Konzeption zu einem Framework für Datenbank-Anwendungen im Web, bei dem ein Rollenkonzept integraler Bestandteil ist und das sich definitionsgetriebene Programmierung zunutze macht.

2.3 Stand der Technik und der Literatur

Die grundlegende Idee dieser Abschlussarbeit – nämlich: Anwendungen weniger durch Codierung und mehr durch Definitionen zu erstellen – wird von zahlreichen Low-Code-Plattformen bereits realisiert.

Für den deutschsprachigen Raum hat sich angeregt durch den *Berlin Low-Code Day 2019* mit der Low-Code Association e. V. ein Verband führender Anbieter von Low-Code Plattformen und Service-Provider gegründet, dem aktuell (Stand Juli 2023) 17 Anbieter angehören [14], die sich in ihrem Manifest überzeugt zeigen, dass „maßgeschneiderte Software … zunehmend mit interaktiven, visuellen und deklarativen Methoden aus vorgefertigten Programmfunktionen zusammengesetzt“ wird [15].

Der Begriff „Low-Code“ wurde 2014 von John Rymer, Vice President und Chief Analyst bei Forrester Research, geprägt und umfasst demnach „Produkte oder Cloud-Dienste für die Anwendungsentwicklung, die statt Programmierung visuelle, deklarative Techniken verwenden“ [16].

Allerdings wird der Begriff „Low-Code“ gemäß Bock und Frank „auf inkonsistente Weise verwendet“ [17]. Entsprechende Produkte werden als „low-code platform“ (LCP), „low-code application platform“ (LCAP) und „low-code development platform“ (LCDP) bezeichnet, wobei der Umfang dieser Produkte zwar voneinander abweicht, jedoch deutliche Überschneidungen erkennen lässt, wie nachstehende Grafik aus [17] sehr anschaulich zeigt.

Gemäß dieser Grafik scheinen Werkzeuge zur Datenmodellierung, zum GUI-Design, zur Rechteverwaltung und zur Bereitstellung der Anwendung wesentliche Aspekte zu sein, die von Low-Code-Plattformen häufig abgedeckt werden.

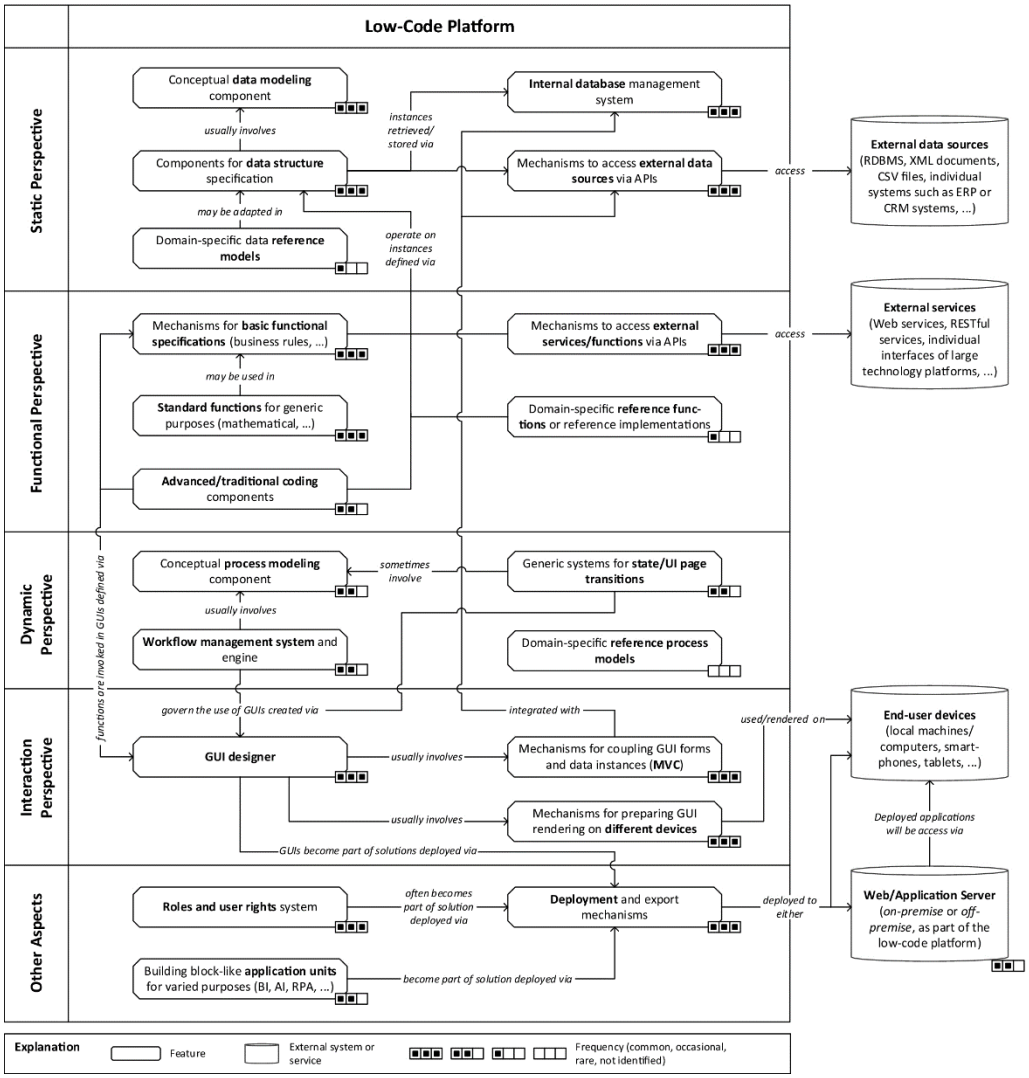


Abbildung 1: Funktionsumfang von LCP; aus Bock und Frank [17]

Mendix, eines der gemäß einem Gartner-Bericht führenden Unternehmen auf diesem Gebiet [18], wirbt beispielsweise damit, von der Einladung der Stakeholder in der Ideenphase bis hin zu Monitoring-Tools für die Betriebsführung den gesamten Lebenszyklus einer Anwendung abzudecken [19].

2.3.1 Vor- und Nachteile von Low-Code Plattformen

Vor- und Nachteile werden in der Folge jeweils gemeinsam betrachtet, da ein als vorteilhaft erachteter Aspekt auch kritisch betrachtet werden kann und soll.

Durchgängige Toolunterstützung

Obige Darstellung des Umfangs von Low-Code Plattformen zeigt bereits einen wesentlichen Vorteil dieser Systeme auf – nämlich die Durchgängigkeit der Werkzeugunterstützung während des Entwicklungsprozesses.

Gerade hierin sehen Bock und Frank jedoch auch die Gefahr der Abhängigkeit von einem Hersteller, da es verheerende Folgen für eine Anwendung hat, wenn der LCP-Hersteller, auf den man gesetzt hat, das System nicht mehr weiterentwickelt, wie dies bei Google App Maker[7] der Fall ist.

Produktivität

Als wesentliches Argument für die Verwendung von Low-Code Plattformen wird regelmäßig die beschleunigte Anwendungsentwicklung angeführt. Dabei wird darauf hingewiesen, dass Low-Code Technologien einen zunehmenden Beitrag dazu leisten können, die digitale Transformation zu ermöglichen [22].

Da es lt. [23] Stand Ende 2022 an wissenschaftlichen Studien mangelt, die die Produktivität von Low-Code Entwicklung der Produktivität von herkömmlicher Entwicklung zuverlässig auf Basis umfangreicher Tests gegenüberstellen, wird in [23] ein Experiment beschrieben, das eine solche Gegenüberstellung in einem überschaubaren Projekt versucht und zu dem Ergebnis kommt, dass die selbe Anwendung mit einem Low-Code System in dreifacher Geschwindigkeit erstellt werden konnte.

Diese Gegenüberstellung ist insofern kritisch zu bewerten, da der Programmumfang und auch die Komplexität des untersuchten Beispiels sehr gering gewählt wurden. Repetitive Aufgaben wie die Umsetzung einfacher CRUD-Operationen lassen sich mit standardisierten Tools natürlich leichter umsetzten als Algorithmen, bei denen ein Großteil des Entwicklungsaufwands nicht im Codieren, sondern in der Lösungsfindung und -optimierung liegt.

Demokratisierung der Softwareentwicklung

Ein weiterer Aspekt, dem in Verbindung mit Low-Code Plattformen Beachtung geschenkt werden sollte, ist die Demokratisierung der Softwareentwicklung [24]. Darunter ist die stärkere Einbeziehung von Nicht-IT-Spezialisten in den Softwareentwicklungsprozess bis hin zur kompletten Umsetzung von IT-Projekten durch sogenannte Citizen Developers gemeint – also Mitarbeiter aus Fachabteilungen, die ihr Know-How der von ihnen bearbeiteten Geschäftsprozesse nutzen, um mit von der IT-Abteilung bereitgestellten Mitteln Anwendungen zu erstellen. Die Auswahl und Betreuung der verwendeten Low-Code Plattformen durch die IT-

[7] App Maker war eine Low-Code Plattform von Google [20], die jedoch am 19.01.2021 abgeschaltet wurde [21]

Abteilung wird dabei als wesentlicher Erfolgsfaktor genannt, um die Entstehung von Schatten-IT zu vermeiden, was nicht zuletzt ein Sicherheitsrisiko wäre [25].

2.3.2 Fazit zu Low-Code Plattformen

Die Zielgruppe der Nicht-IT-Spezialisten macht eine durchgängige Werkzeugunterstützung aus einem Guss erforderlich. Dabei finden sich jedoch keine grundsätzlich neuen Ansätze in den Low-Code Produkten. Visuelle Werkzeuge zum GUI-Design, zur Datenmodellierung oder zur Geschäftsprozessmodellierung traten nicht erst mit Low-Code Plattformen in Erscheinung und stellen somit auch kein Alleinstellungsmerkmal von ihnen dar. Cabot behauptet sogar: „I do not believe there is any fundamental technical contribution in low-code trend“ [26].

2.3.3 Abgrenzung zu bestehenden Low-Code Plattformen

Dieses Dokument und das hier beschriebene Framework beschränken sich auf den Teil des Software-Entwicklungsprozesses, bei dem traditionell tatsächlich codiert wird.

Relevante Teile der o. g. Low-Code Plattformen haben nicht im engeren Sinne mit Codierung zu tun bzw. lösen diese nicht ab. Die in Low-Code Plattformen gebotene Werkzeugunterstützung für Anforderungsmanagement, Datenmodellierung oder Betriebsführung reduziert nicht direkt den Codier-Aufwand.

Daher sind diese Aspekte nicht Gegenstand der weiteren Erörterungen. Stattdessen geht es hier ausschließlich um diejenige Gemeinsamkeit zu Low-Code Plattformen, die das Schreiben wiederkehrenden Programmcodes reduziert.

Das hier vorgestellte Framework richtet sich auch ausdrücklich nicht an Citizen Developers, sondern an Software-Entwicklerinnen, die mittels des gezielten Einsatzes von Definitionen den Umfang ihrer Codier-Arbeit reduzieren möchten.

Hier steht eine wiederkehrende Aufgabenstellung von Datenbankanwendungen – nämlich die Realisierung von CRUD-Operationen – im Vordergrund und es wird gezeigt, wie diese mit wenig Codierung realisiert werden kann. Die hier dargestellte Vorgehensweise lässt sich in die herkömmliche Programmierung integrieren und erfordert keinen kompletten Umstieg auf eine Low-Code Plattform, so dass die oben beschriebene Herstellerabhängigkeit vermieden wird.

Im Fokus stehen Definitionen, die die Funktionsweise einer Anwendung beschreiben, und die Art wie sie von einem Framework ausgewertet werden können. Die Definitionen werden hier in einer Form vorgestellt, wie sie mit einem einfachen Texteditor bearbeitet werden können, wenngleich die Erstellung visueller

Werkzeuge zur grafischen Bearbeitung solcher Definitionen eine denkbare Ausbaustufe wäre.

Die Auswahl der hier betrachteten Definitionen ergibt sich aus dem Anwendungsgebiet, für das das Framework konzipiert wird – nämlich Datenbank-Anwendungen im Web, bei denen ein Rollenkonzept darüber bestimmt, wer welche Daten wie bearbeiten darf.

3 Definitionstypen

3.1 Komposition einer Anwendung

3.1.1 Zweck

Eine Web-Anwendung setzt sich typischerweise aus mehreren GUI-Komponenten zusammen, wobei das Menü eine zentrale GUI-Komponente darstellt, mittels derer der Anwender Programmfunktionen aufrufen kann. Weitere Komponenten visualisieren die zu bearbeitenden Daten.

Ziel des hier vorgestellten Frameworks für Web-Anwendungen (im weiteren Verlauf „das Framework“ genannt) besteht darin, typische für eine Anwendung benötigte Komponenten aus Definitionen zu generieren, so dass die Anwendungsentwicklerin diese Komponenten nicht programmiert, sondern die Anwendungen an Hand von Definitionen zusammensetzt. Mit diesem Framework ist kein bestimmtes Framework gemeint, auch wenn es wie eingangs beschrieben bereits ein solches gibt, sondern es ist ein nach den hier beschriebenen Anforderungen erstelltes oder zu erstellendes Framework gemeint.

Eine mit diesem Framework zusammengesetzte Anwendung basiert vornehmlich auf drei Arten von Definitionen, die im Folgenden näher erläutert werden.

1. Menü-Definitionen
2. Abfrage-Definitionen
3. Filter-Definitionen

Diese Definitionen legen fest, welche Daten wie dargestellt werden sollen. Das Framework wertet diese Definitionen aus und präsentiert dem Anwender eine Anwendung, deren Menü sich aus den Menü-Definitionen ergibt und deren Menüpunkte die in Abfrage-Definitionen angegebenen Daten zeigen, wobei Filter-Definitionen diese Daten rollenspezifisch eingrenzen.

Es ist somit eine Anforderung an das Framework, die nachstehend beschriebenen Definitionen auszuwerten. Diese Arbeit beschreibt hierbei, getreu der deklarativen Herangehensweise, was definiert werden können muss und was hierbei die Erwartung an das interpretierende Framework ist. Wie das Framework die Interpretation der Definitionen realisiert, ist nicht Bestandteil dieser Arbeit.

Der Aufbau und die Funktionalität der Anwendung werden durch Definitionsdateien festgelegt.

3.1.2 Beispielanwendung

Die in diesem Papier beschriebenen Konzepte werden zum besseren Verständnis und zur Vergleichbarkeit des Schreibaufwands gegenüber der herkömmlichen Programmierung an Hand von Beispielen beschrieben. Diese ergeben eine Beispielanwendung, die u. a. den Funktionsumfang der Bibliotheksverwaltung aus [2] umfasst. Daher wird auch das physische Datenmodell[8] der Beispielanwendung aus [2] übernommen. Die dort verwendeten drei Tabellen und eine weitere werden in nachstehendem Datenmodell dargestellt und in der folgenden Tabelle kurz beschrieben.

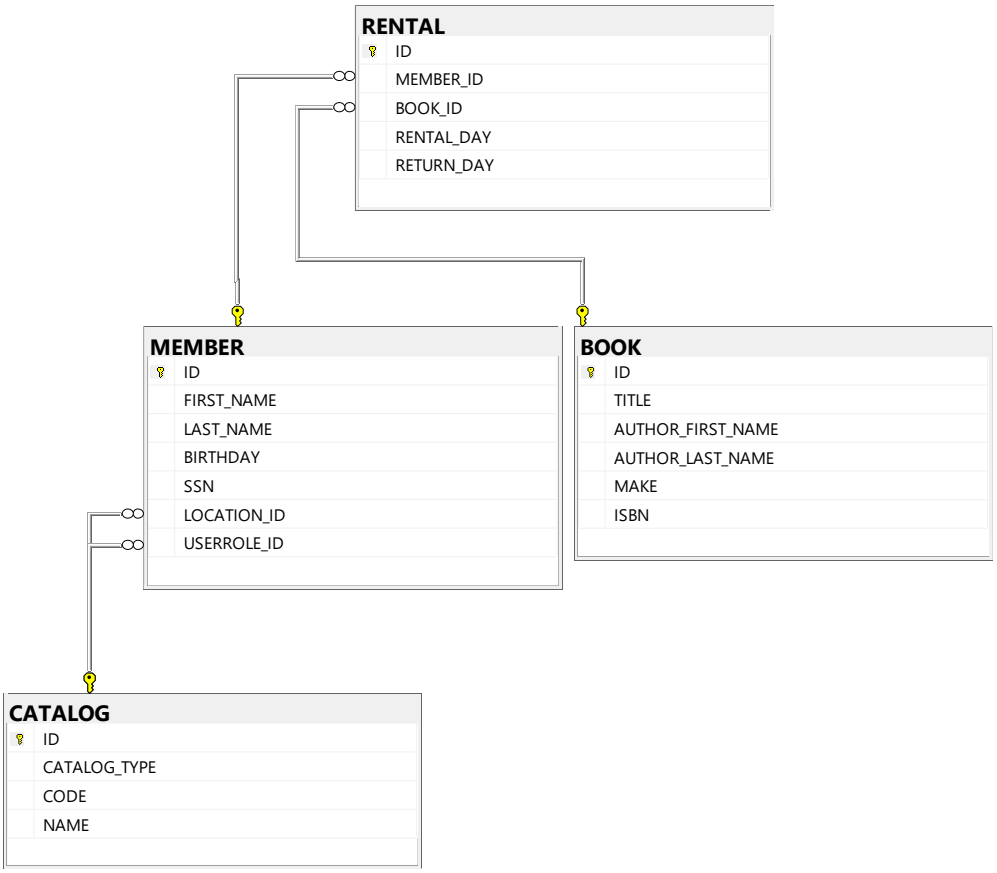


Abbildung 2: Datenmodell der Beispielanwendung in Anlehnung an [2]

[8] Ein physisches Datenmodell geht aus einem konzeptionellen hervor und berücksichtigt technische Aspekte wie Schlüsselspalten. In einem physischen Datenmodell gibt es Tabellen, in denen konkrete Objekte als Zeilen abgelegt werden sollen. In einem konzeptionellen Datenmodell gibt es Entitätstypen, die Entitäten beschreiben. In diesem Papier wird daher von Entitätstypen und Entitäten gesprochen, wenn es ausdrücklich um konzeptionelle Überlegungen geht, während im Falle konkreter Definitionsbeispiele von Tabellen und Zeilen bzw. Datensätzen die Rede ist.

Tabelle	**Bedeutung in [2] und hier vorgenommene Erweiterungen**
MEMBER	Benutzer der Bibliothek In der hier nachimplementierten Variante, die zur Demonstration weiterführender Konzepte über die in [2] beschriebene Funktionalität hinausgeht, können alle Benutzer der Bibliothek gleichzeitig Benutzer der Web-Anwendung sein. Daher wurde die Tabelle um zwei Fremdschlüssel auf Kataloge mit Rollen und Standorten erweitert.
BOOK	In der Bibliothek angebotene Buchtitel Es handelt sich wie in [2] nicht um konkrete Bücher mit einem Regalplatz oder Ausleihstatus, sondern nur um die angebotenen Titel mit minimalen Attributen
RENTAL	Ausleihen Hier wird festgehalten, wer wann welches Buch entliehen hat. Zusätzlich zu [2] wurde hier auch das Rückgabedatum aufgenommen
CATALOG	Allgemeingültige, nicht in [2] enthaltene Katalog-Tabelle, in der sämtliche Kataloge abgebildet werden können (hier Standorte und Rollen). Statt für jeden Katalog eine eigene Tabelle zu erstellen, wird eine allgemeingültige Tabelle bevorzugt, in der der CATALOG_TYPE angibt, zu welchem Katalog die jeweiligen Zeilen gehören.

Tabelle 1: Tabellenbeschreibung der Beispielanwendung

Die Primärschlüssel sämtlicher Tabellen heißen ID. Alle Spalten sind Pflichtfelder bis auf das Rückgabedatum (RETURN_DAY) einer Ausleihe.

3.1.3 Live-Demo

Zur Demonstration wurde die hier vorgestellte Beispielanwendung namens Bux unter http://bux.ismaiel.de bereitgestellt.

Die Anmeldung ist u. a. mit dem Administrator-Login *ADM-000002* und dem Kennwort *Rainer* möglich. Alle weiteren Benutzer kann man im Menüpunkt

Stammdaten/Benutzer einsehen. Das Attribut Benutzer-Nr. dient als Login-Name und der Vorname als Kennwort. Dies ist freilich sehr unsicher, ist jedoch darin begründet, dass zur besseren Vergleichbarkeit möglichst wenige Ergänzungen zum Datenmodell von [2] vorgenommen wurden.

Die mit der Beispielanwendung aus [2] überschneidende Funktionalität von Bux befindet sich ausschließlich hinter den Menüpunkten *Stammdaten* und *Vorgänge*. Die übrige Funktionalität der Beispielanwendung dient der Darstellung weiterer GUI-Komponenten, ihrer durch Definitionen festgelegten Interaktion miteinander und der Verdeutlichung des Rollenkonzepts.

Für die Beispieldatenbank wurden knapp 6.000 Buchtitel dem Katalog der Deutschen Nationalbibliothek entnommen. Rund 17.000 Benutzerdatensätze und 70.000 Ausleihen wurden zufällig generiert.

Zur Verdeutlichung des in diesem Framework implementierten Rollenkonzepts sind die Benutzer auf 19 Standorte verteilt und haben eine der drei Rollen *Administrator*, *Mitarbeiter* und *Mitglied* und verfügen damit über die in Tabelle 2 genannten Berechtigungen.

Rolle	**Berechtigung**
ADMIN Administrator	Darf alle Daten sehen und bearbeiten
EMPLOYEE Mitarbeiter	Darf alle Daten seines Standorts sehen und bearbeiten; d. h. die Benutzer seines Standorts und deren Ausleihen. Er darf zwar alle Bücher sehen, aber keine bearbeiten.
MEMBER Benutzer	Darf nur seine Daten sehen und nicht bearbeiten. Er darf zwar alle Bücher sehen, aber keine bearbeiten.

Tabelle 2: Rollen der Beispielanwendung

Die Datenbanktabelle *MEMBER* enthält also nicht nur Mitglieder, sondern alle Akteure der Bibliothek. Diese können gleichzeitig auch Anwender des Systems sein.

3.1.4 Aufbau

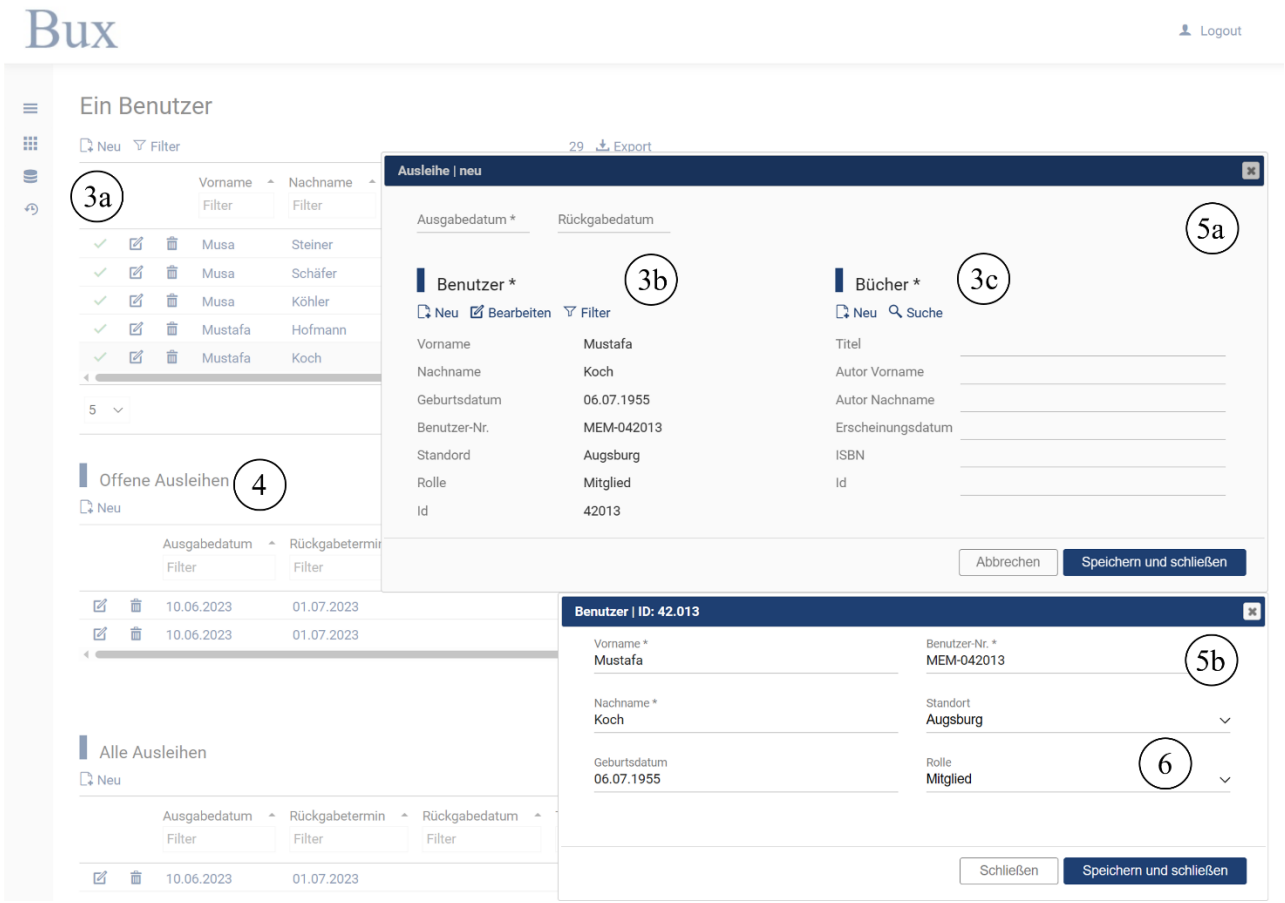
Die Benutzerführung setzt sich aus verschiedenen GUI-Komponenten zusammen.

Das hier vorgestellte Framework unterscheidet zwischen strukturierenden und datenselektierenden GUI-Komponenten, wobei die strukturierenden den Aufbau, die Anordnung und die Hierarchie der GUI-Komponenten festlegen, während die datenselektierenden GUI-Komponenten Daten anzeigen, die beim Backend angefragt werden.

Nachstehende beiden Abbildungen zeigen Ansichten innerhalb der Beispielanwendung, in der sämtliche GUI-Komponenten (unterhalb des Seitenkopfes) an Hand von Definitionen aufgebaut wurden.

Strukturierende GUI-Komponenten sind dabei mit römischen und datenselektierende GUI-Komponenten sind mit arabischen Zahlen durchnummeriert.

Screenshot 1: Menü und ein Dashboard der Beispielanwendung



Screenshot 2: Dashboard mit zwei Editoren

Die nachstehende Tabelle nennt und beschreibt, sofern nicht selbsterklärend, kurz die dargestellten GUI-Komponenten.

Nr.	**Beschreibung**
I	Menü
II	Dashboard
III	Box innerhalb eines Dashboards
IV	Karteireiterleiste
V	Karteireiter
1	KPI, Key Performance Indicator, Kennzahl, auch Kennzahl genannt
2	Business-Grafik

Nr.	Beschreibung
	a) Tortendiagramm b) Balkendiagramm Weitere Diagrammtypen sind denkbar
3	SDT, **S**ingle Row **D**ata **T**able Spezielle Art von Tabelle zur Auswahl eines einzelnen Datensatzes 3a) Übersichtsmodus zur Auswahl aus einer Menge 3b) Einzelsatzdarstellung nach Auswahl eines Eintrags 3c) Suchmodus zur Eingabe von Suchkriterien SDTs können u. a. verwendet werden, um in einer Fremdschlüsselbeziehung Daten des referenzierten Objektes anzuzeigen – beispielsweise Details eines Benutzers oder Buches in einem Editor zur Bearbeitung einer Ausleihe, in der je ein Benutzer und ein Buch referenziert werden (siehe Editor 5b der obigen Abbildung).
4	Tabelle
5	Editor
6	Kombinationslistenfeld, auch Combobox oder Dropdown genannt

Tabelle 3: Kurzbeschreibungen der GUI-Komponenten

3.1.5 Auswertung

Eine mit dem hier beschriebenen Framework erstellte Anwendung setzt sich aus den Komponenten zusammen, deren hierarchische Struktur und Darstellungsform innerhalb der Menü-Definition der jeweiligen Rolle festgelegt und vom Client entsprechend ausgewertet wird.

Für datenselektierende GUI-Komponenten ist zudem eine Interaktion mit dem Server erforderlich, der die Daten für die jeweilige Komponente liefert, wobei dies unabhängig von der Darstellungsform erfolgt. Welche Daten der Server liefern soll, wird in Abfrage-Definitionen festgelegt.

3.1.6 Anforderungen an das Framework

Es muss mittels Definition festgelegt werden können,

1. wie einzelne Komponenten zueinander angeordnet werden sollen,
2. wie einzelne Komponenten dargestellt werden sollen und
3. welche Daten für eine Komponente geliefert werden sollen.
4. Es müssen mindestens die in *Tabelle 3* genannten Darstellungsformen unterstützt werden.
5. Die Liste der unterstützten Darstellungsformen muss möglichst leicht erweiterbar sein (die Erstellung einer neuen Darstellungsform erfordert freilich Codierung; die Nutzung einer neuen Darstellungsform dagegen nicht).

Zur ausführlichen Beschreibung der hier vorgestellten datenselektierenden GUI-Komponenten wird auf den *Anhang A: Datenselektierende GUI-Komponenten* verwiesen.

3.2 Menü-Definitionen

3.2.1 Zweck

Eine Menü-Definition definiert das Menü einer Benutzer-Rolle. Anwender mit der selben Rolle erhalten somit das selbe Menü. Die Zuordnung, welche Benutzer-Rolle welche Menü-Definition verwendet, ist über eine Dateinamenskonvention zu regeln – etwa Rollenname zzgl. Dateinamenserweiterung für Menüs.

Definitionen werden in Definitions-Dateien abgelegt. Das Framework verfügt über eine hinreichende Konfigurierbarkeit, so dass der Speicherort der unterschiedlichen Definitionstypen einstellbar ist.

Neben der Hierarchie und der Bezeichnung der einzelnen Menüpunkte wird in einer Menü-Definition auch festgelegt, was bei Auswahl eines Menüpunkts geschehen soll.

Ziel der Verwendung von Menü-Definitionen ist es also, an Stelle der fest verdrahteten Programmierung eines Menüs das Menü zur Laufzeit aus der Menü-Definition abzuleiten und so flexibel Anpassungen und Erweiterungen vornehmen zu können und unterschiedlichen Rollen unterschiedliche Menüs bereitzustellen.

Neben der Benutzerführung erfüllen Menü-Definitionen eine wichtige Funktion des Rollen-Konzepts, da sie festlegen, welche Rolle Zugriff auf welche Funktionalität und damit auf welche Daten hat. Nachstehende Grafik stellt die betrachteten Datenbestände auf der Abszisse und die Rollen auf der Ordinaten dar.

Eine weitere Dimension, also praktisch die Applikante, ergibt sich aus den Operationen, die Anwender einer Rolle pro Datenbestand anwenden dürfen – hier vereinfacht mit „lesen“ und „bearbeiten“ wiedergegeben.

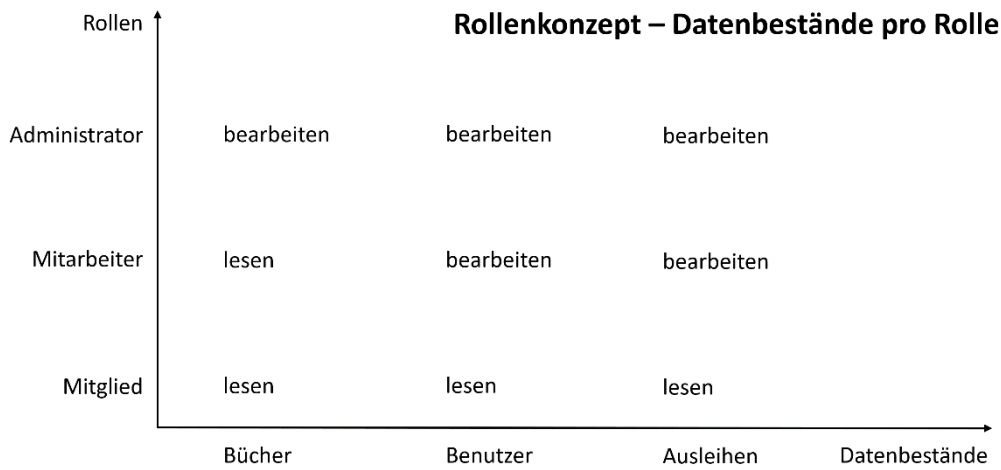


Abbildung 3: Rollenkonzept – Datenbestände pro Rolle

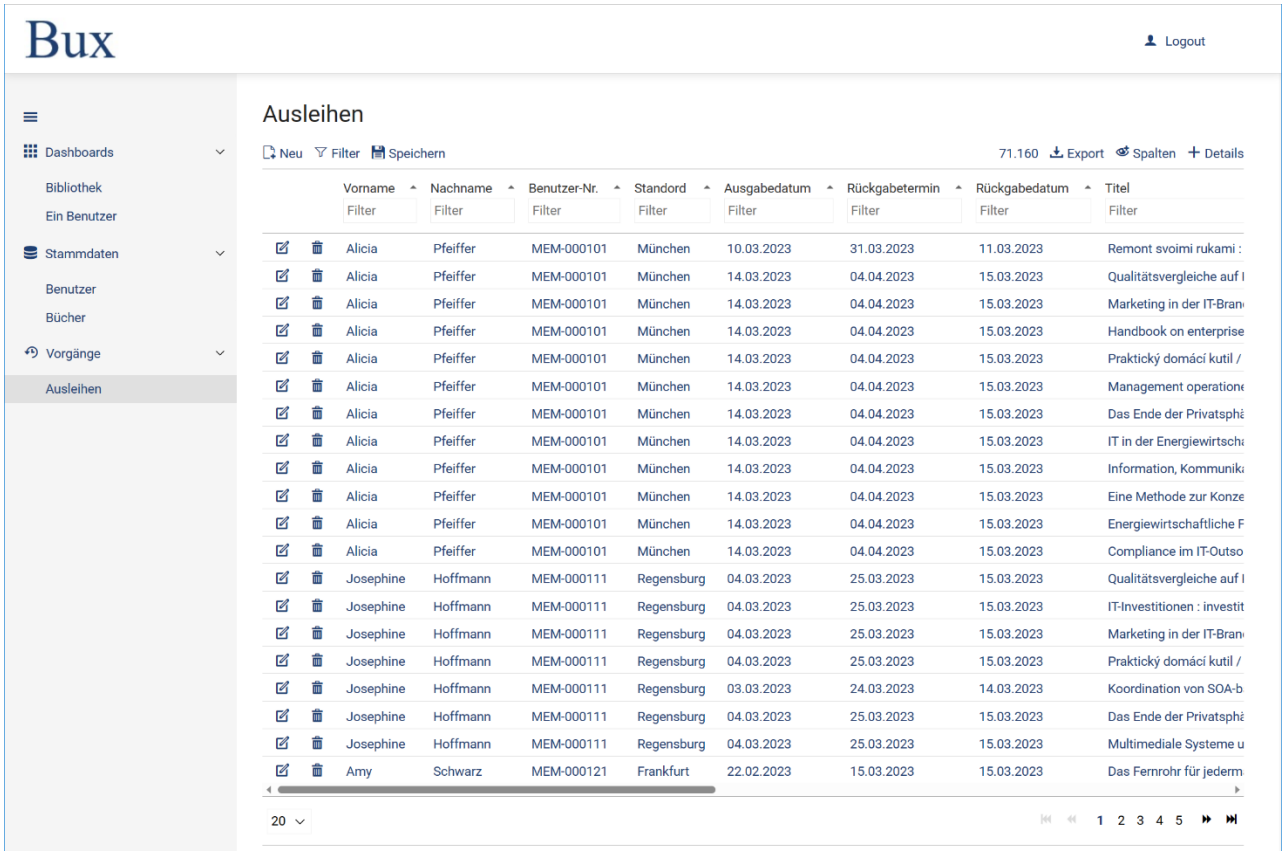
3.2.2 Aufbau

Menü-Definitions-Dateien haben die Struktur eines Arrays in JavaScript Object Notation (JSON). Jedes Array-Element stellt dabei einen Menüpunkt dar.

Eine einfache Menü-Definition für die Rolle *ADMIN*, aus der das in Screenshot 3 abgebildete Menü resultiert, könnte beispielsweise wie in Definitionsbeispiel 1 aussehen.

```
[ { "Id":"MENU_COLLAPSE",      "Type":"function", "Icon":"icon-menu7"
  , "Function":"window.parent.udqMenuManager.toggleMenuMode();"
  , "Class":"menu-button"
  }
, { "Id":"DASHBOARD",          "Type":"menu",  "Label":"Dashboards"
  , "Icon":"icon-grid",        "_children":
    [ { "Id":"OVERVIEW_DASH",  "Type":"dash",  "Label":"Bibliothek"
      , "Autostart":"yes",     "Include":"SUB_OVERVIEW_DASH.menu"
      }
    , { "Id":"MEMBER_DASH",    "Type":"dash",  "Label":"Ein Benutzer"
      , "Columns":"2",         "Include":"SUB_MEMBER_DASH.menu"
      }
    ]
  }
, { "Id":"STAMMDATEN",         "Type":"menu",  "Label":"Stammdaten"
  , "Icon":"icon-database",    "_children":
    [ { "Id":"MEMBER",         "Type":"table", "Label":"Benutzer"
      , "File":"MEMBER.query", "CRUD":"CRUD"
      }
    , { "Id":"BOOK",           "Type":"table", "Label":"Bücher"
      , "File":"BOOK.query",   "CRUD":"CRUD"
      }
    ]
  }
, { "Id":"VORGANG",            "Type":"menu",  "Label":"Vorgänge"
  , "Icon":"icon-history",     "_children":
    [ { "Id":"RENTAL",         "Type":"table", "Label":"Ausleihen"
      , "File":"RENTAL.query", "CRUD":"CRUD"
      }
    ]
  }
, { "Id":"VERSTECKT",          "Type":"menu",  "Label":"Programminterna"
  , "CRUD":"H",                "Include":"SUB_INTERNAL.menu"
  }
]
```

Definitionsbeispiel 1: Menü-Definitionsdatei ADMIN.menu



Screenshot 3: Hauptseite mit Menüstreifen links und Arbeitsbereich rechts

Auf der obersten Menü-Ebene befinden sich die drei Menüpunkte *Dashboards*, *Stammdaten* und *Vorgänge*. Der Menüpunkt *Dashboards* beinhaltet in seinem Attribut _children ein Array mit zwei Untermenüpunkten für *Bibliothek* und *Ein Benutzer*, die Übersichten über den gesamten Datenbestand (Dashboard *Bibliothek*) bzw. zu einem bestimmten Benutzer (Dashboard *Ein Benutzer*) zeigen.

Analog besitzen die Menüpunkte *Stammdaten* und *Vorgänge* Untermenüpunkte zur Auflistung und Bearbeitung von *Büchern*, *Benutzern* und *Ausleihen*.

Ein weiterer Menüpunkt beinhaltet Definitionen zu programminterner Funktionalität, die nicht direkt aus dem Menü heraus aufgerufen werden kann, aber ebenso über die Rolle (bzw. deren Menü-Definition) freigeschaltet wird.

Das Framework präsentiert dem Anwender auf Grund von obiger Menü-Definition das in *Screenshot 3* abgebildete Menü, das von der Bezeichnung der Menüpunkte und der Hierarchie der obigen JSON-Struktur entspricht, was u. a. an dem Attribut *Label* der einzelnen Menü-Elemente zu erkennen ist.

3.2.3 Auswertung

Nach der Anmeldung fragt das Frontend des Frameworks die Menü-Definition für den angemeldeten Anwender am Backend des Frameworks an, interpretiert diese bei Erhalt und stellt dem Anwender ein entsprechendes Menü dar. Dazu enthält die Hauptseite der Web-Anwendung einen Menü-Streifen auf der linken Seite, in dem das Menü angezeigt wird.

Das Layout der Haupt-Seite inkl. Header-Bereich mit Logo und Logout-Schaltfläche sowie die Anmelde-Seite und der Wechsel auf die Haupt-Seite nach erfolgreicher Anmeldung werden hier nicht näher betrachtet, da sie nicht zum Schwerpunkt dieser Arbeit gehören. Es wird jedoch vorausgesetzt, dass es eine solche Haupt-Seite gibt, die einen Bereich für das Menü und einen „Arbeitsbereich" genannten Teil besitzt, in dem später konkrete Inhalte gezeigt werden.

Backendseitig wird ein Endpunkt benötigt, der die Menü-Definitions-Datei der Rolle des jeweiligen Anwenders vom Dateisystem liest und an den Client liefert, wobei die Rolle als Attribut im JWT-Token enthalten sein kann.

3.2.4 Attribute

Obige Notation eines Menüs lässt beliebig viele Hierarchieebenen zu, wobei sich übergeordnete Menüpunkte dadurch auszeichnen, dass sie ein Attribut *_children* mit einer Liste von untergeordneten Menüpunkten haben. Ansonsten besitzen Menüpunkte unabhängig von ihrer Hierarchieebene die selben Attribute.

Nachstehende Tabelle nennt und beschreibt kurz die im Verlauf dieses Dokuments näher beschriebenen Attribute einer Menü-Definition.

Attribut	**Bedeutung**
Id	Eindeutiger Bezeichner des Menüpunktes Dieser wird benötigt, um zur Laufzeit Berechtigungen abfragen zu können – etwa, indem geprüft wird, ob der jeweilige Anwender auf Grund seiner Rolle über einen bestimmten Menüpunkt verfügt.
Label	Bezeichnung des Menüpunktes aus Nutzersicht Dies ist die Bezeichnung, die dem Anwender in der GUI angezeigt wird.

Attribut	**Bedeutung**
Type	Typ des Menüpunktes Je nach Darstellungsform des durch den Menüpunkt aufgerufenen Inhalts haben Menüpunkte unterschiedliche Typen, die in *Tabelle 3* genannt und im Abschnitt *Anhang A: Datenselektierende GUI-Komponenten* näher beschrieben werden.
File	Definitionsdatei für die jeweilige Aktion Da nicht nur der Aufbau des Menüs, sondern auch die dadurch ausgelösten Aktionen definitionsgetrieben sind, kann pro Menüpunkt eine Definitionsdatei angegeben werden. An Stelle einer Datei kann die Aktion auch anderweitig festgelegt werden, wie z. B. bei dem ersten Menüpunkt, der einfach eine JavaScript-Funktion ausführt, die den Menüstreifen auf- und zuklappt.
Function	JavaScript-Ausdruck, der bei Auswahl des Menüpunkts ausgeführt wird (alternativ zu *File*)
Icon	Anzuzeigendes Symbol pro Menüpunkt
CRUD	Erlaubte Operationen für den jeweiligen Menüpunkt Die Operationen werden in Form einer Zeichenfolge angegeben, die die Anfangsbuchstaben der jeweiligen Operationsbezeichnungen enthält: C =CREATE: Neuanlage R = READ: Lesen U = UPDATE: Ändern D = DELETE: Löschen H = HIDDEN: versteckter Menüpunkt F = FORBIDDEN: verbotener Menüpunkt

Attribut	**Bedeutung**
	Falls nicht angegeben, gilt R
_children	Liste mit Untermenüpunkten
Include	Eingebundene Menü-Definitions-Datei. Dies dient zum Einen dazu, Menü-Definitionen nicht unübersichtlich groß werden zu lassen, und zum anderen dazu, dass man Blöcke von Menüpunkten in mehreren Menü-Definitionen unterschiedlicher Rollen wiederverwenden kann.
Columns	Anzahl der Spalten in einem Menü-Element des Typs *box*
Column	Position eines Menü-Elements innerhalb eines Menü-Elements des Typs *box*
Description	Array mit den Zeilen des Beschreibungstextes für ein Info-Icon, der beim Herüberfahren mit der Maus angezeigt wird
Parameters	Array mit Key/Value-Paaren, die als Parameter an eine Abfrage-Definition übergeben werden, um dortige Platzhalter (Key) mit entsprechenden Werten (Value) zu ersetzen
Class	CSS-Klasse zur Darstellung des Eintrags im Menü
FilterCriteria	Angabe, ob Suchkriterien abgefragt werden sollen Wird nur für Tabellen und SDTs verwendet; wenn die Angabe nicht explizit auf *no* gesetzt wird, werden Suchkriterien angezeigt.

Tabelle 4: Attribute einer Menü-Definition

3.2.5 Anforderungen an das Framework

Hinsichtlich der Menü-Definitionen ergeben sich somit folgende Anforderungen an das Framework:

1. Das Frontend muss die Menü-Definition einer Rolle abfragen können
2. Das Backend muss die Menü-Definition einer Rolle liefern können
3. Das Frontend muss die Menü-Definition als Menü darstellen können

Vorausgesetzt und hier nicht weiter beschrieben ist der Anmelde-Vorgang bei Programmstart und wie dabei die Rolle des angemeldeten Anwenders ermittelt wird. Des Weiteren muss die Rolle bei der Abfrage der Menü-Definition backendseitig aus den Stammdaten des Anwenders gelesen werden.

3.3 Abfrage-Definitionen

3.3.1 Zweck

Eine Abfrage-Definition legt fest, was bei Auswahl eines Menüpunktes geschehen soll. Hierzu kann ein Menüpunkt, wie im Abschnitt zu *Menü-Definitionen* bereits beschrieben, über ein Attribut *File* verfügen, das die Datei der zum Menüpunkt gehörigen Abfrage-Definition angibt.

Aus einer Abfrage-Definition werden backendseitig zur Laufzeit Datenbank-Statements generiert, die die gewünschten Daten liefern, speichern oder löschen. Ferner kann eine Abfrage-Definition Informationen zur Darstellung am Frontend enthalten.

Ziel ist es also, das Programmverhalten zu einem Menüpunkt sowohl backend- als auch frontendseitig nicht fest verdrahtet zu codieren, sondern aus einer Abfrage-Definition abzuleiten.

Bei Auswahl des Menüpunktes *Ausleihen* aus obigem Beispiel soll eine Tabelle (`"Type":"table"`) angezeigt werden (siehe *Screenshot 3*), deren Inhalt sich nach der im Attribut *File* angegebenen Abfrage-Definition richtet (`"File":"RENTAL.query"`).

Die Abfrage-Definition gibt an, welche Daten selektiert werden sollen und wie sie dargestellt werden (einige Angaben wurden hier zur besseren Übersicht herausgekürzt).

3.3.2 Aufbau

Auch Abfrage-Definitionen werden als Objekt im JSON-Format angegeben. Sie besitzen in ihrer Minimal-Form ein Attribut namens *Columns* mit einem Array von Spalten-Definitionen und ein Attribut namens *Tables* mit einem Array von Tabellen-Definitionen, die angeben, aus welchen Tabellen die Daten entnommen werden sollen und wie diese miteinander verknüpft sind.

Weitere Attribute der Abfrage-Definition wie *Filters* und *Orders* ergänzen die Abfrage um eine Liste von konjunktiv verknüpften Filter-Bedingungen und Angaben zur Sortierung. Beide Attribute sind Arrays mit Strings.

```
{ "Columns":
  [ ...
  , { "Table":"MEMBER",      "Name":"FIRST_NAME",       "Alias":"FIRST_NAME"
    , "Type":"string",       "Group":"Benutzer",        "Label":"Vorname"
    }
  , { "Table":"MEMBER",      "Name":"LAST_NAME",        "Alias":"LAST_NAME"
    , "Type":"string",       "Group":"Benutzer",        "Label":"Nachname"
    }
  , ...
  , { "Table":"RENTAL",      "Name":"RENTAL_DAY",       "Alias":"RENTAL_DAY"
    , "Type":"date",         "Group":"Ausleihe",        "Label":"Ausgabedatum"
    }
  , { "Table":"",   "Name":"RENTAL.RENTAL_DAY + 21",    "Alias":"DUEDATE"
    , "Type":"date",         "Group":"Ausleihe",        "Label":"Rückgabetermin"
    }
  , { "Table":"RENTAL",      "Name":"RETURN_DAY",       "Alias":"RETURN_DAY"
    , "Type":"date",         "Group":"Ausleihe",        "Label":"Rückgabedatum"
    }
  , { "Table":"BOOK",        "Name":"TITLE",           "Alias":"TITLE"
    , "Type":"string",       "Group":"Buch",            "Label":"Titel"
    }
  , { "Table":"BOOK",        "Name":"AUTHOR_LAST_NAME"
    , "Alias":"AUTHOR_LAST_NAME"
    , "Type":"string",       "Group":"Buch",            "Label":"Autor"
    }
  , { "Table":"BOOK",        "Name":"ISBN",            "Alias":"ISBN"
    , "Type":"string",       "Group":"Buch",            "Label":"ISBN"
    , "width":400
    }
  , { "Table":"RENTAL",      "Name":"ID",              "Alias":"ID"
    , "Type":"number",       "Group":"Ausleihe",        "Label":"Id"
    , "Constraint":"PK",     "Serial":"AUTO"
    }
  , ...
  ]
, "Tables":
  [ { "Name":"RENTAL"}
  , { "Name":"MEMBER",  "Alias":"MEMBER",   "JoinType":"JOIN"
    , "JoinCondition":"MEMBER.ID   = RENTAL.MEMBER_ID"
    }
  , { "Name":"CATALOG", "Alias":"LOCATION", "JoinType":"JOIN"
    , "JoinCondition":"LOCATION.ID = MEMBER.LOCATION_ID"
    }
  , { "Name":"BOOK",    "Alias":"BOOK",     "JoinType":"JOIN"
    , "JoinCondition":"BOOK.ID     = RENTAL.BOOK_ID"
    }
  ]
, "Orders":
  [ "RENTAL.RENTAL_DAY DESC"
  , "MEMBER.LAST_NAME"
  ]
}
```

Definitionsbeispiel 2: Abfrage-Definition RENTAL.query

3.3.3 Auswertung

Aus einer solchen Definition kann das Backend ein SQL-Statement generieren, das die angegebenen Spalten aus den ebenfalls angegebenen und entsprechend den Angaben zu verknüpfenden Tabellen selektiert.

Das SELECT-Statement, das an Hand von obiger Abfrage-Definition generiert wird, sieht folgendermaßen aus:

```
SELECT MEMBER.FIRST_NAME      AS [FIRST_NAME]
     , MEMBER.LAST_NAME       AS [LAST_NAME]
     , MEMBER.SSN             AS [SSN]
     , LOCATION.NAME          AS [LOCATION]
     , RENTAL.RENTAL_DAY      AS [RENTAL_DAY]
     , RENTAL.RENTAL_DAY + 21 AS [DUEDATE]
     , RENTAL.RETURN_DAY      AS [RETURN_DAY]
     , BOOK.TITLE             AS [TITLE]
     , BOOK.AUTHOR_LAST_NAME  AS [AUTHOR_LAST_NAME]
     , BOOK.ISBN              AS [ISBN]
     , RENTAL.ID              AS [ID]
     , RENTAL.MEMBER_ID       AS [MEMBER_ID]
     , RENTAL.BOOK_ID         AS [BOOK_ID]
  FROM RENTAL
  JOIN MEMBER MEMBER      ON (MEMBER.ID   = RENTAL.MEMBER_ID)
  JOIN CATALOG LOCATION   ON (LOCATION.ID = MEMBER.LOCATION_ID)
  JOIN BOOK BOOK          ON (BOOK.ID     = RENTAL.BOOK_ID)
 ORDER BY
       RENTAL.RENTAL_DAY DESC
     , MEMBER.LAST_NAME
```

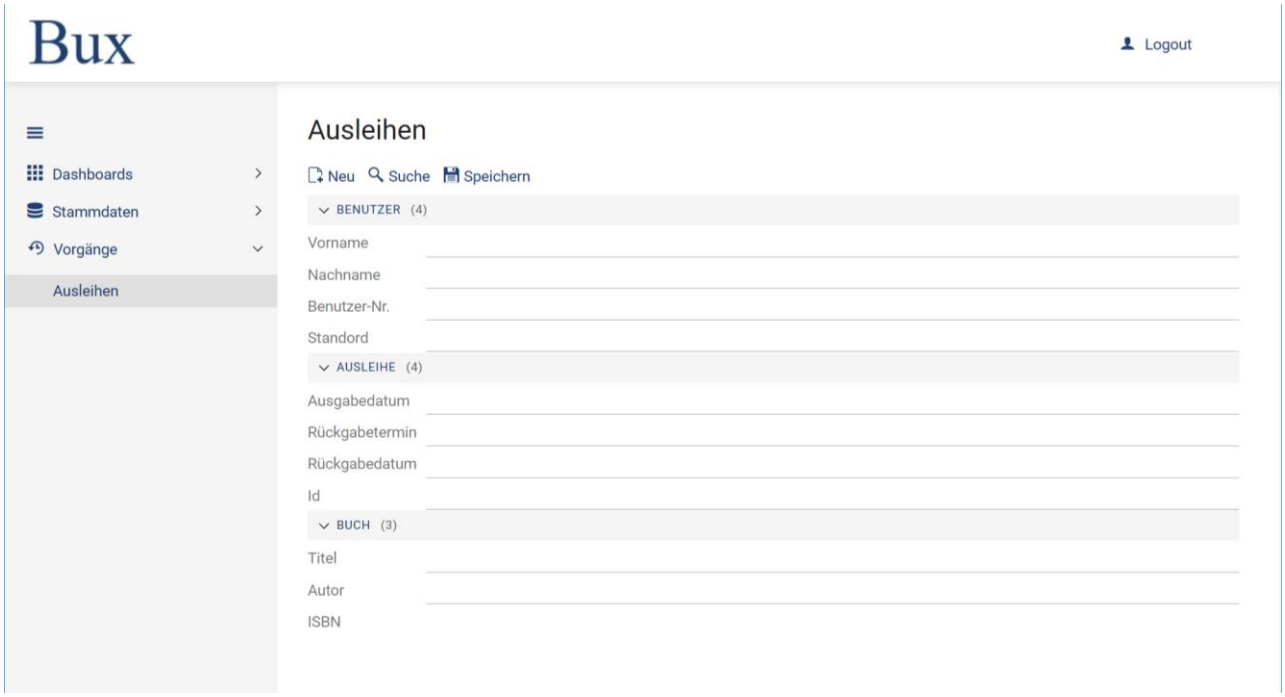
Statement 1: generiertes Select-Statement

Es ist direkt ersichtlich, wie aus der Abfrage-Definition ein Select-Statement generiert wird.

Theoretisch ließe sich eine Abfrage-Definition auch direkt als Select-Statement hinterlegen, doch die Abfrage-Definition dient nicht nur der Selektion, sondern auf ihrer Basis sollen auch weitere DML-Statements generiert werden können.

Des Weiteren ermöglicht die Abstraktion von SQL an dieser Stelle auch die Anreicherung um weitere, nicht SQL-relevante Attribute wie z. B. das Label, das in der Abfrage-Definition pro Spalte angegeben wird, um die Spaltenüberschrift in der GUI-Tabelle festzulegen, die die selektierten Daten darstellen soll.

Das selbe Label kann auch verwendet werden, um der Anzeige des Suchergebnisses einen Suchkriterien-Dialog vorzuschalten, in dem die selben Aufschriften für die einzelnen Suchkriterien verwendet werden.



Screenshot 4: vorgeschalteter Suchkriterien-Dialog

Dies bedeutet, dass Abfrage-Definitionen sowohl vom Backend als auch vom Frontend verwendet werden:

Das Backend generiert die Datenbank-Statements an Hand der Tabellen- und Spalten-Listen und ggf. weiteren Attributen zur Sortierung, Filterung und Gruppierung; und das Frontend baut das Layout entsprechend den *Label*- und *Type*-Eigenschaften auf.

3.3.4 Attribute

Nachstehende Tabelle nennt und beschreibt kurz die im Verlauf dieses Dokuments näher beschriebenen Attribute einer Abfrage-Definition. Zu Attributen, für deren Verständnis über dieses Kapitel hinausgehende Erklärungen erforderlich sind, ist jeweils das Kapitel angegeben, in dem sie näher beschreiben werden.

Attribut	**Bedeutung**
Columns	Liste mit Spalten-Definitionen (Array mit Objekten), aus der die Select-Liste eines SELECT-Statements generiert wird Pro Element der Select-Liste kann es folgende Attribute geben: • Table: Tabelle bzw. Alias, aus der das Element selektiert wird

Attribut	**Bedeutung**
	• Name: Spaltenname oder Ausdruck, der selektiert werden soll • Alias: Alias, mit dem der Ausdruck aliasiert wird • Type: grundlegender Datentyp (string, number, date) • Label: Beschriftung in der GUI • Group: Gruppierung in der GUI • Icon: Anzuzeigendes Icon • formatter: Formatierungsanweisung z. B. „money“ für Betragsfelder (weitere können implementiert werden; bzgl. dieses und der fünf folgenden Attribute siehe: *Tabellen*) • bottomCalc: Anzeige des Werts einer Aggregatfunktion (sum, avg, min, max) im Gruppen oder Tabellen-Fuß • ShowTable: in Tabellenansicht zeigen • ShowDetail: in Detailansicht zeigen • Filter: als Filterkriterium anbieten • Constraint: Angabe, wenn es sich um einen Primär- oder Fremdschlüssel handelt (siehe *Einzelsatzdarstellung* und *Darstellung von untergeordneten Objekten*) • Serial: Verfahren zur PK-Generierung (siehe *Abfrage-Definition des Editors*) • Button: Anzeige einer Schaltfläche an Stelle eines selektierten Wertes; mögliche Ausprägungen: edit: Bearbeiten-Schaltfläche, wenn die Berechtigung besteht, sonst verstecken editOrView: Bearbeiten-Schaltfläche, wenn die Berechtigung besteht, sonst Anzeige-Schaltfläche anzeigen delete: Lösch-Schaltfläche, wenn die Berechtigung besteht, sonst verstecken select: Schaltfläche zur Datensatzauswahl

Attribut	**Bedeutung**
	(bzgl. dieses und der beiden folgenden Attribute siehe *Schaltflächen* und *Einzelsatzdarstellung*) • LabelColumns: Spalten, die für eine Lösch-Warnung verwendet werden sollen • Condition: Bedingung, falls eine Spalte nur unter bestimmten Umständen angezeigt werden soll
Tables	Liste mit Tabellen-Definitionen (Array mit Objekten), aus der die FROM-Clause inkl. JOINs generiert wird. Pro Element dieser Liste kann es folgende Attribute geben: • Table: Name der Datenbanktabelle • Alias: Alias, mit der sie aliasiert werden soll • JoinType: Art und weise, wie gejoint werden soll (JOIN, LEFT OUTER JOIN, OUTER APPLY, CROSS APPLY etc.) • JoinCondition: Bedingung, mit der gejoint werden soll
Filters	Liste von Filterbedingungen (Array mit Strings)
Orders	Liste von Sortierkriterien (Array mit Strings)
Groups	Liste von Gruppierungskriterien (Array mit Strings)
Values	Liste mit Key/Value-Paaren für weiterführende Funktionalität (z. B. zur Angabe des zu verwendenden Editors für die Bearbeitung eines Datensatzes – siehe: *Editoren*)
Options	Angaben zu Drilldowns – siehe *Drilldown*
TableOptions	Frontendseitige Optionen zur Gruppierung (siehe: *Tabellen*)

Tabelle 5: Attribute einer Abfrage-Definition

3.3.5 Vorteile von Abfrage-Definitionen

Die Anwendungs-Entwicklerin braucht sich bei der Verwendung eines solchen Frameworks nicht um technische Aspekte wie die Übermittlung des Requests inkl. der eingegebenen Suchkriterien, die Auswertung der Suchkriterien und die Erstellung und Absetzung des SQL-Statements, die Entgegennahme der Daten vom Datenbankserver, die Serialisierung und Übertragung der Daten an den Client etc. zu kümmern. Sie braucht lediglich festzulegen, welche Daten selektiert werden sollen, und muss dazu natürlich das Datenmodell kennen und wissen, wie die Tabellen miteinander zu verknüpfen sind.

Die Benutzerführung kann ihr ebenfalls durch das Framework abgenommen werden, indem grundsätzlich aus der Definition ein Suchkriterien-Dialog generiert und angeboten wird, beim Klicken auf die Schaltfläche *Suchen* ein Request abgesetzt wird und bei Entgegennahme der Daten eine Tabellenansicht generiert und angezeigt wird. Die generierten Elemente resultieren nicht in generiertem Quellcode, sondern werden zur Laufzeit aus der Definition erstellt und verwendet.

Diese Vorgehensweise sorgt zudem für eine einheitliche Benutzerführung, da das Framework gleichartige Menüpunkte immer in der selben Art behandelt.

Ein weiterer Vorteil einer solchen Vorgehensweise ist, dass Komfortfunktionen wie die Filterung über Filterfelder im Spaltentitel, eine Detail- oder Einzelsatzdarstellung beim Hovern[9] über eine Datenzeile, der Export (z. B. nach Excel), die Vorbelegung von Suchkriterien, das Ein- und Ausblenden von Spalten oder die Änderung ihrer Reihenfolge und das anwenderspezifische Abspeichern solcher Einstellungen u. s. w. nur einmal im Frontend-Teil des Frameworks implementiert werden müssen und fortan für jede Abfrage-Definition in gleicher Weise zur Verfügung stehen.

Es existiert in obigem Beispiel keine fest implementierte GUI für Bücher, Benutzer oder Ausleihen, sondern lediglich ein frontendseitiger Interpreter für Abfrage-Definitionen. Zusätzlich hinzugefügte Komfortfunktionen müssen somit ebenfalls nicht in jeden Menüpunkt einzeln eingefügt werden.

In Verbindung mit Menü-Definitionen ist bereits jetzt erkennbar, was zu tun ist, wenn die Anwendung weitere Inhalte zeigen können soll. Neben den drei Abfrage-Definitionen für Benutzer, Bücher und Ausleihen, die den Funktionsumfang der

[9] Als Hovern wird das Rüberfahren mit dem Mauszeiger über ein Bedienelement bezeichnet.

Beispielanwendung in [2] definieren, sind für die Beispielanwendung ähnliche Abfrage-Definitionen für die Tabellen und Grafiken auf den in *Screenshot 2* und *Screenshot 3* vorgestellten Dashboards erstellt worden.

Wenn also beispielsweise unterhalb des Menüpunktes *Stammdaten* zusätzlich noch eine Liste der Standorte oder Rollen angezeigt werden sollte, wird jeweils lediglich ein weiterer Eintrag in der Menü-Definition benötigt, in dem eine Abfrage-Definition angegeben wird, die besagt, aus welchen Tabellen welche Spalten selektiert werden sollen – sprich: eine Definitions-Datei wird ergänzt und eine wird neu erstellt. Die Anwendung selbst, die bisher im Wesentlichen aus dem Framework besteht, kann unverändert bleiben.

Es liegt auf der Hand, dass eine solche Änderung mit geringem Testaufwand ausgeliefert werden kann. Die geänderte Menü-Definition kann fehlerhaft sein, indem sie z. B. kein gültiges JSON-Format aufweist. Dies würde dazu führen, dass das Menü nicht dargestellt werden kann. Hierzu ist also vor der Auslieferung einer neuen Version ein Test erforderlich.

Eine zusätzliche Abfrage-Definitions-Datei für Standorte oder Rollen würde jedoch, falls sie fehlerhaft wäre, lediglich eine Fehlfunktion in der neuen Funktionalität, d. h. der Anzeige von Standorten oder Rollen, aufweisen. Die bestehende Funktionalität bliebe davon unberührt.

Das Framework verwendet zur Darstellung von Tabellen ein geeignetes JavaScript-Widget. Das vom Autor implementierte Framework nutzt Tabulator [27], da es sehr mächtig und äußerst gut dokumentiert ist. Grundsätzlich kann ein nach den hier beschriebenen Grundsätzen implementiertes Framework jedoch auch ein anderes Widget verwenden. Es sollte jedoch eine Adapter-Klasse zwischengeschaltet werden, statt direkt Methoden und Eigenschaften des Widgets zu verwenden. Das zu verwendende Widget muss die Fähigkeit besitzen, Spalten zur Laufzeit erzeugen zu können, da die benötigten Spalten erst durch das Lesen der Definition bekannt werden, und beliebige, zur jeweiligen Spaltenmenge passende Ergebnismengen darzustellen, die in Form eines JSON-Arrays entgegengenommen werden.

3.3.6 Anforderungen an das Framework

Bezüglich Abfrage-Definitionen ergeben sich somit folgende Anforderungen an das Framework:

1. Das Frontend muss bei Auswahl eines Menüpunktes die betreffende Abfrage-Definition abfragen können.
2. Das Backend muss die angeforderte Abfrage-Definition liefern können.
3. Das Frontend muss an Hand der Abfrage-Definition im Arbeitsbereich der Oberfläche entsprechende Bedienelemente erstellen können.
4. Zunächst muss das Frontend Eingabefelder für Suchkriterien und eine Suchschaltfläche darstellen können.
5. Bei Betätigung der Suchschaltfläche muss das Frontend die eingegebenen Suchkriterien an das Backend weiterleiten und die Daten anfordern.
6. Das Backend muss die Suchkriterien auswerten, ein aus der Abfrage-Definition abgeleitetes SELECT-Statement generieren, an den Datenbankserver senden und die von diesem erhaltenen Daten als Antwort auf die im vorgenannten Punkt genannte Anforderung an den Client liefern können.
7. Das Frontend muss die gelieferten Daten entsprechend dem *Type* der jeweiligen datenselektierenden GUI-Komponente darstellen können.

3.4 Filter-Definitionen

3.4.1 Zweck

Obige Ausführungen zeigen, wie man definitionsgetrieben die Bandbreite der dargestellten Daten festlegen kann, indem pro Rolle eine entsprechende Menü-Definition festgelegt wird. Die Menü-Definition benennt also die erlaubten Entitätstypen (z. B. Benutzer, Bücher, Ausleihen).

Darüber hinaus ist es erforderlich, pro Rolle innerhalb eines solchen Entitätstyps festlegen zu können, welche Entitäten ein Anwender einer bestimmten Rolle sehen darf – etwa nur die Benutzer eines bestimmten Standorts. Dies ist vergleichbar zur Row-Level-Security beim Microsoft SQL Server [28]. (Zur Begründung, weshalb hierfür nicht Row-Level-Security verwendet wird, siehe Abschnitt *Rollen-Filter vs. Row-Level-Security* im Anhang)

Ziel der Filter-Definitionen ist es also, pro Kombination aus Rolle und Datenbanktabelle (und ggf. auch Kontext, was im Abschnitt *Wiederverwendung von Filter-Definitionen* beschrieben wird) Bedingungen festlegen zu können, die regeln, wer auf welche Zeilen der Tabelle zugreifen darf.

3.4.2 Vorbereitende Überlegungen und Maßnahmen

Für die nachstehende Betrachtung wird auf die Rollen *EMPLOYEE* und *MEMBER* der Beispielanwendung zurückgegriffen, wobei Anwender der Rolle *EMPLOYEE* nur die Daten ihres Standorts sehen dürfen – sprich: Benutzer ihres Standorts und Ausleihen von Benutzern ihres Standorts. Die Tabelle *MEMBER* erfüllt in diesem Beispiel eine doppelte Funktion. Zum einen stehen dort die Benutzerdaten der Anwender – pro Anwender der Anwendung befindet sich dort ein Datensatz, in dem u. a. auch sein Standort angegeben ist. Zum anderen können alle dort verzeichneten Anwender gleichzeitig auch Benutzer der Bibliothek und einem Standort zugeordnet sein.

Wenn in den folgenden Ausführungen von einem Datensatz der Tabelle *MEMBER* im Sinne eines Anwenders der Anwendung die Rede die Rede ist, wird der Begriff „Anwender“ verwendet; ist dagegen von einem Benutzer der Bibliothek die Rede, wird von einem „Benutzer“ gesprochen.

Anwender der Rolle *EMPLOYEE* sehen Daten von Benutzern, die den selben Standort haben wie sie selbst. Sie sollen auch nur Ausleihen dieser Benutzer sehen und nicht diejenigen von Benutzern anderer Standorte.

Anwender der Rolle *MEMBER* dürfen nur ihre eigenen Daten sehen – also ihren eigenen Benutzer-Datensatz und ihre Ausleihen.

Es wird zunächst davon ausgegangen, dass beide Rollen *EMPLOYEE* und *MEMBER* das selbe Menü verwenden wie die Rolle *ADMIN*. Wie dies ohne Duplizieren von Definitionen erreicht werden kann, wird im Abschnitt *Wiederverwendung und Überladung* beschrieben.

Durch Filter-Definitionen erhält das Rollen-Konzept eine weitere Dimension. Während die Unterscheidung, welche Daten eine Rolle sehen oder bearbeiten darf, eine horizontale Dimension darstellt, kommt nun innerhalb eines Datenbestands eine vertikale Unterscheidung hinzu, die regelt, wessen Daten ein Anwender einer bestimmten Rolle sehen oder bearbeiten darf.

Um dies zu erreichen, wirken Filter-Definitionen für je eine Rolle auf je eine Tabelle. Durch eine Filter-Definition beschränkt man also für Anwender der Rolle *EMPLOYEE* die Sichtbarkeit der Datensätze der Tabelle *MEMBER* auf diejenigen Benutzer, die den selben Standort haben, wie der jeweilige Anwender dieser Rolle.

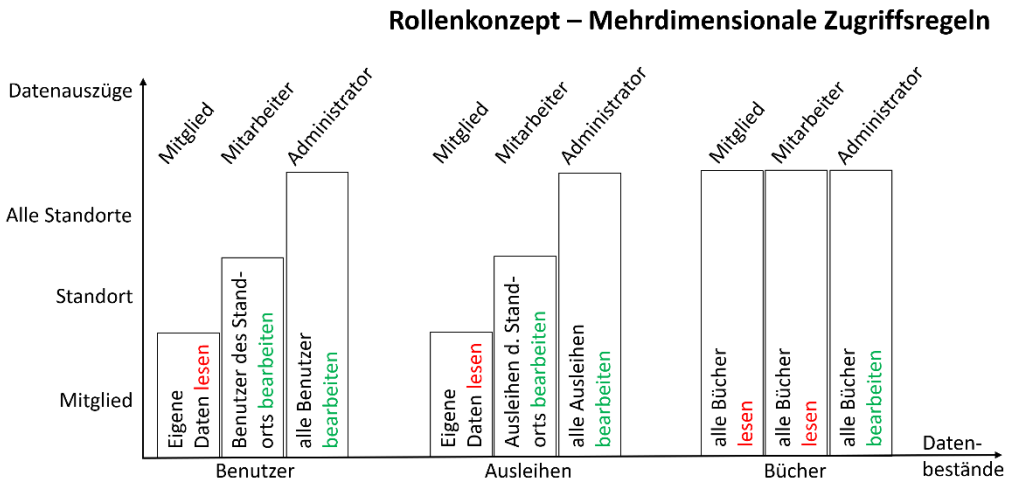


Abbildung 4: Vertikale Differenzierung der Rollen

3.4.3 Aufbau

Eine Filter-Definition wirkt immer dann, wenn ein Anwender der jeweiligen Rolle auf Daten der jeweiligen Tabelle zugreift.

Die Dateinamenskonvention für Filter-Definitionen besteht daher aus dem Namen der Rolle, für die sie definiert wird, dem Namen der Datenbanktabelle, auf die sie als Zeilenfilter einwirken soll, und einer Dateinamenserweiterung (jeweils getrennt durch einen Punkt). Um die Zeilen der Tabelle *MEMBER* so zu filtern, dass Anwender der Rolle *EMPLOYEE* nur diejenigen Benutzer ihres Standorts

sehen dürfen, bedarf es also einer Filter-Definitionsdatei mit dem Dateinamen *EMPLOYEE.MEMBER.query*.

Eine Filter-Definition ist eine Abfrage-Definition, die als EXISTS-Clause an ein auf Basis einer Abfrage-Definition generiertes SELECT-Statement angehängt wird. Die Abfrage-Definition bestimmt das Haupt-Statement und für sämtliche darin enthaltene Tabellen kann es Filter-Definitionen geben, die jeweils EXISTS-Clauses definieren, mit denen die jeweils betroffene Tabelle gefiltert wird.

Die Spalten-Liste einer solchen Filter-Definition enthält nur einen Eintrag, der auch keine Datenbankspalte benennt, sondern einfach nur ein beliebiges Literal (z. B. 1), da diese Abfrage keine Daten liefert, sondern nur für eine EXISTS-Clause benötigt wird.

Die Tabellen-Liste einer Filter-Definition zeigt den Weg auf von der jeweiligen Zeile in der Anwender-Tabelle (in betrachtetem Beispiel die Tabelle *MEMBER*, die die Anmeldedaten sämtlicher Anwender der Anwendung enthält) hin zu der für die Filterung maßgeblichen Tabelle, die das Merkmal enthält, das die Filterung bestimmt (in betrachtetem Beispiel ebenfalls die Tabelle *MEMBER*, die in diesem Falle die zu filternden Objekte – nämlich Benutzer – enthält).

Da Filter-Definitionen spezielle Abfrage-Definitionen sind, können sie wie jede andere Abfrage-Definition ein Attribut *Filters* mit einer Liste von Bedingungen enthalten. Diese Liste enthält für das oben umrissene Beispiel eine Bedingung, die die Tabelle mit den Login-Daten (also die Tabelle *MEMBER*) auf den aktuellen Anwender einschränkt, und eine Bedingung, die die für den Filter maßgebliche Tabelle mit der betroffenen Tabelle aus dem Haupt-Statement verknüpft. Beide Bedingungen verwenden Platzhalter, die zur Laufzeit entsprechend ersetzt werden – zum einen der Platzhalter *#<username>#*, der durch den Login-Namen des angemeldeten Anwenders ersetzt wird, und der Platzhalter *#<PARENT>#*, der durch den Alias ersetzt wird, mit dem die betroffene Tabelle im Haupt-Statement aliasiert ist bzw. ihr eigentlicher Name, wenn sie nicht aliasiert wird.

```
1  { "Columns":
2    [ { "Table":"", "Name": "1", "Type":"number", "Label":"ID"}
3    ]
4  , "Tables":
5    [ { "Name":"MEMBER", "Alias":"USR_MEMBER"}
6    ]
7  , "Filters":
8    [ "USR_MEMBER.SSN           = '#<username>#'"
9    , "USR_MEMBER.LOCATION_ID   = #<PARENT>#.LOCATION_ID"
10   ]
11 }
```

Definitionsbeispiel 3: Filter-Definition EMPLOYEE.MEMBER.query

3.4.4 Auswertung

Im Falle der oben beschriebenen Abfrage-Definition *RENTAL.query* wirkt die obige Filter-Definition namens *EMPLOYEE.MEMBER.query* auf die in der Abfrage-Definition angegebene Tabelle *MEMBER*, wenn diese Abfrage-Definition von einem Anwender der Rolle *EMPLOYEE* verwendet wird.

Anwender der Rolle *EMPLOYEE* und solche der Rolle *ADMIN* verfügen beide über den Menüpunkt *Ausleihen*, der die Abfrage-Definition *RENTAL.query* verwendet, aber nur bei Anwendern der Rolle *EMPLOYEE* wird bei der Statement-Generierung eine EXISTS-Clause angefügt, die sich aus der Filter-Definition *EMPLOYEE.MEMBER.query* ableitet. Dadurch sehen Anwender der Rolle *EMPLOYEE* nur einen Ausschnitt der Tabelle *MEMBER*, während Anwender der Rolle *ADMIN* alle Zeilen der Tabelle *MEMBER* sehen dürfen (es sei denn, es gäbe eine Filter-Definition namens *ADMIN.MEMBER.query*, die eine entsprechende Einschränkung für die Rolle *ADMIN* definiert).

Bei der Statement-Generierung wird an das aus der Abfrage-Definition generierte Haupt-Statement eine EXISTS-Clause angehängt, die die dortige Tabelle *MEMBER* über deren *LOCATION_ID* mit der als *USR_MEMBER* aliasierten *MEMBER*-Tabelle des Sub-Selects verknüpft. Man vergleiche hierzu das nachfolgende Statement mit dem, das im Abschnitt *Abfrage-Definitionen* zu dieser Abfrage-Definitionsdatei abgebildet ist und sich lediglich dadurch unterscheidet, dass dort die EXISTS-Clause fehlt, da es oben für die Rolle *ADMIN* ausgeführt wurde, für die es keine Filter-Definition auf die Tabelle *MEMBER* gibt.

Die bloße Existenz einer Filter-Definition für die Kombination aus Rolle und eine in einer Abfrage-Definition enthaltenen Tabelle sorgt also dafür, dass eine Zeilenfilterung für die jeweilige Tabelle stattfindet. Hierzu bedarf es natürlich einer geeigneten Dateinamenskonvention und eines definierten (bzw. konfigurierbaren) Ablageortes für Filter-Definitionen.

Das Vorhandensein dieser Filter-Definition greift also automatisch sowohl bei dem hier beschriebenen Menüpunkt *Ausleihen* als auch bei dem Menüpunkt *Benutzer* ein, wo ebenfalls die Tabelle *MEMBER* verwendet wird. Des gleichen wirkt sie in sämtlichen Abfragen der beiden Dashboards – sprich: es ist zentral festgelegt, wieviel Anwender der Rolle *EMPLOYEE* aus der Tabelle *MEMBER* sehen dürfen; und dies wirkt dann in sämtlichen Abfragen mit dieser Tabelle.

```
1  SELECT MEMBER.FIRST_NAME      AS [FIRST_NAME]
2       , MEMBER.LAST_NAME       AS [LAST_NAME]
3       , MEMBER.SSN             AS [SSN]
4       , LOCATION.NAME          AS [LOCATION]
5       , RENTAL.RENTAL_DAY      AS [RENTAL_DAY]
6       , RENTAL.RENTAL_DAY + 21 AS [DUEDATE]
7       , RENTAL.RETURN_DAY      AS [RETURN_DAY]
8       , BOOK.TITLE             AS [TITLE]
9       , BOOK.AUTHOR_LAST_NAME  AS [AUTHOR_LAST_NAME]
10      , BOOK.ISBN              AS [ISBN]
11      , RENTAL.ID              AS [ID]
12      , RENTAL.MEMBER_ID       AS [MEMBER_ID]
13      , RENTAL.BOOK_ID         AS [BOOK_ID]
14    FROM RENTAL
15    JOIN MEMBER MEMBER      ON (MEMBER.ID   = RENTAL.MEMBER_ID)
16    JOIN CATALOG LOCATION   ON (LOCATION.ID = MEMBER.LOCATION_ID)
17    JOIN BOOK BOOK          ON (BOOK.ID     = RENTAL.BOOK_ID)
18   WHERE ( EXISTS
19           ( SELECT 1 AS [F0]
20               FROM MEMBER USR_MEMBER
21              WHERE ( USR_MEMBER.SSN          = 'EMP-000020')
22                AND ( USR_MEMBER.LOCATION_ID  = MEMBER.LOCATION_ID )
23           )
24         )
25   ORDER BY
26         RENTAL.RENTAL_DAY DESC
27       , MEMBER.LAST_NAME
```

Definitionsbeispiel 4: generiertes Select-Statement mit Filter

In diesem Beispiel wurde von dem der Rolle *EMPLOYEE* zugeordneten Anwender *EMP-000020* der Menüpunkt *Ausleihen* aufgerufen. Das Sub-Select der EXISTS-Clause beschränkt seine Ergebnismenge zunächst auf die *MEMBER*-Zeile des Anwenders mit dem Login ‚*EMP-000020*' und anschließend werden die Benutzer (Tabelle *MEMBER*) des Haupt-Statements auf diejenigen eingeschränkt, die zum selben Standort gehören wie der angemeldete Anwender.

Um auch für Anwender der Rolle *MEMBER* den Sichtbarkeitsbereich auf die Daten des jeweiligen Anwenders einzuschränken, ist lediglich eine weitere Filter-Definition namens *MEMBER.MEMBER.query* anzulegen, bei der die Bedingung in Zeile 9 der obigen Filterbedingung ersetzt wird durch eine Filterbedingung, die nicht auf den selben Standort, sondern auf die selbe Benutzer-Id einschränkt – also: `"USR_MEMBER.ID  = #<PARENT>#.ID"`

Damit ist bereits die vertikale Dimension des in *Grafik 3* dargestellten Rollenkonzepts abgeschlossen.

3.4.5 Anforderungen an das Framework

Filter-Definitionen betreffend ergeben sich somit folgende Anforderungen an das Framework:

1. Das Backend muss bei der Statement-Generierung für jede in einer Abfrage-Definition enthaltene Tabelle prüfen, ob es dazu für die Rolle des betroffenen Anwenders eine Filter-Definition gibt.
2. Sofern es sie gibt, muss aus der Filter-Definition eine EXISTS-Clause generiert und an das Statement angehängt werden – bei Tabellen, die mit einem INNER JOIN angehängt werden, kann dies in der WHERE-Clause des Hauptstatements erfolgen; bei mit OUTER JOIN verknüpften Tabellen muss dies in der JOIN-Bedingung erfolgen. Der Filter würde dabei nicht die Ergebnismenge einschränken, sondern nur die Sichtbarkeit der Attribute der gefilterten Tabelle.

4 Schreibende Zugriffe

Die bisherigen Ausführungen beschränkten sich auf das Lesen von Daten. Abfrage-Definitionen dienen dabei dem Backend zur Generierung eines SELECT-Statements.

Desgleichen kann das Backend jedoch auch UPDATE-, INSERT- und DELETE-Statements aus einer Abfrage-Definition generieren.

Was hierzu definiert werden können muss, soll in der Folge dargelegt werden.

4.1 Schaltflächen

4.1.1 Zweck

Für die drei typischen Bearbeitungsfunktionen Neuanlage, Bearbeitung und Löschen werden dem Benutzer entsprechende Schaltflächen zur Verfügung gestellt.

4.1.2 Aufbau

Typischerweise erfolgt der Zugriff auf zu bearbeitende Datensätze über eine Auswahlliste (hier Browser genannt) für entsprechende Objekte. Als Browser eignen sich die bereits vorgestellten Tabellendarstellungen, die zu Bearbeitungszwecken mit entsprechenden Schaltflächen versehen werden – eine Neu-Schaltfläche im Tabellenkopf und Bearbeiten- und Lösch-Schaltflächen pro Zeile (siehe *Screenshot 3*).

4.1.3 Auswertung

Die Schaltflächen sollen nur sichtbar sein, wenn der Anwender durch seine Rolle die Berechtigung besitzt, Datensätze des jeweiligen Typs zu bearbeiten, neu anzulegen oder zu löschen.

Ferner dürfen die Endpunkte, die bei Betätigung der Schaltflächen aufgerufen werden, serverseitig nur bei Vorliegen der entsprechenden Berechtigungen ausgeführt werden, damit ein Anwender nicht durch Ausführung von JavaScript (etwa in der Konsole der Entwicklertools seines Internet-Browsers) die Endpunkte direkt (also ohne etwaige Schaltflächen) ausführt.

Zur Definition solcher Berechtigungen sieht das Framework vor, einem Menüpunkt das Attribut *CRUD* (für Create, Read, Update, Delete) zu geben, das als Wert

eine Auflistung der Berechtigungen in Form ihrer Anfangsbuchstaben enthalten kann (siehe *Tabelle 4* im Abschnitt *Attribute* des Kapitels *Menü-Definitionen*).

Wenn das Attribut *CRUD* nicht angegeben ist, besteht definitionsgemäß das Leserecht.

Die Angabe *CRUD* wird vom Backend verwendet, um sicherzustellen, dass nur berechtigte Anwender entsprechende Endpunkte aufrufen können.

Das Frontend kann diese nutzen, um entsprechende Schaltflächen anzuzeigen. Während die Neu-Schaltfläche oberhalb einer Tabelle des Browsers direkt vom Frontend entsprechend dem Vorhandensein der Create-Berechtigung erstellt werden kann, ist es für die Anzeige der Schaltflächen zum Editieren und Löschen erforderlich, diese Berechtigungen in der Abfrage-Definition auszuwerten, um entsprechende Spalten in den Zeilen der Tabelle des Browsers darzustellen.

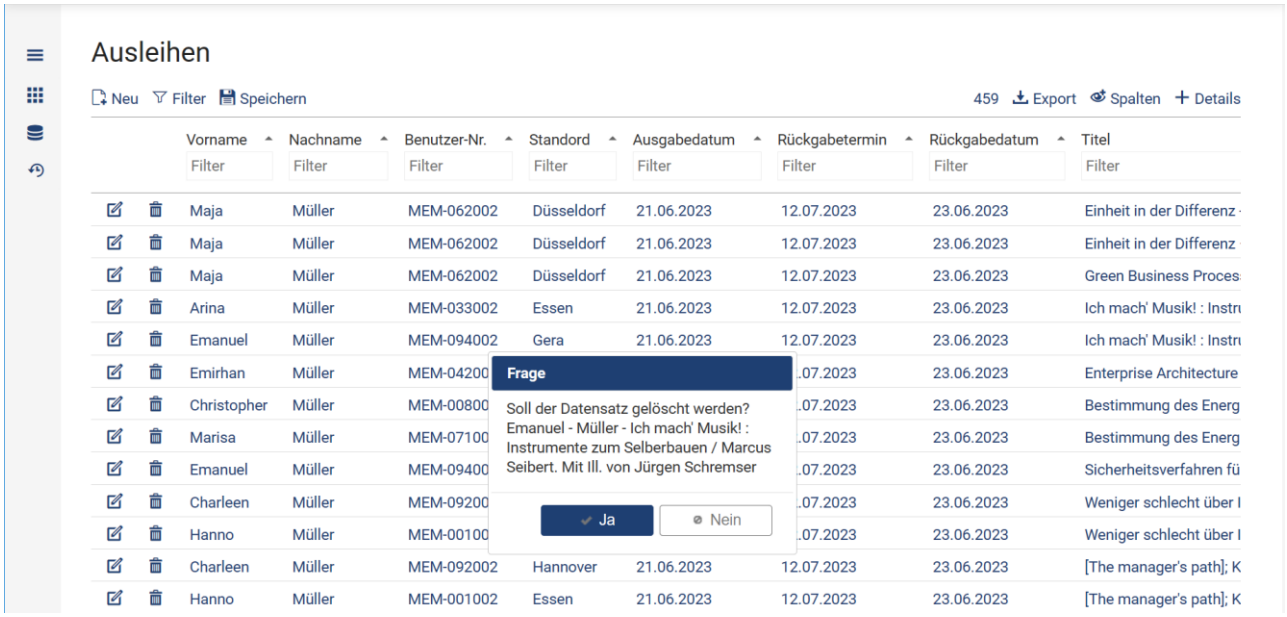
Die Abfrage-Definition enthält zu diesem Zweck in ihrer Spalten-Liste zwei Spalten-Definitionen, die keine Daten selektieren, sondern eine Schaltfläche darstellen (siehe Zeilen 3 und 4 des nachstehenden Ausschnitts aus der Abfrage-Definition *RENTAL.query*).

```
1 { "Columns":
2   [ ...
3   , { "Button":"editOrView"}
4   , { "Button":"delete"
5     , "LabelColumns":"'MEMBER.FIRST_NAME', 'MEMBER.LAST_NAME', 'BOOK.TITLE'"}
6   , { "Table":"MEMBER",      "Name":"FIRST_NAME",       "Alias":"FIRST_NAME"
7     , "Type":"string",       "Group":"Benutzer",        "Label":"Vorname"
8     , "Condition":"!QueryArea.hasMaster()"
9     }
```

Definitionsbeispiel 5: Definitionen für Schaltflächen in RENTAL.query

4.1.4 Löschen

Im Falle der Lösch-Schaltfläche wurde in diesem Beispiel ein Button vom Typ *delete* angegeben, den das Framework nur bei entsprechender Berechtigung anzeigt. Er erfordert eine Angabe der Spalten, die zur Anzeige der Sicherheitsabfrage verwendet werden.



Screenshot 5: Sicherheitsabfrage vor dem Löschen

Wird die Sicherheitsabfrage bestätigt, wird ein entsprechender Request an das Backend gesendet, der die Löschung vornimmt. Im Erfolgsfall wird die entsprechende Zeile aus der Browser-Tabelle entfernt. Andernfalls erscheint eine entsprechende Fehlermeldung.

Das entsprechende DELETE-Statement generiert das Backend aus der Abfrage-Definition, wobei die dortige Angabe zur Primärschlüssel-Spalte sowie die vom Frontend mitgegebene Id des zu löschenden Datensatzes ausgewertet wird.

```
DELETE FROM RENTAL WHERE (ID = 168416)
```

Statement 2: Generiertes Lösch-Statement

4.1.5 Editieren

Für das Editieren wird in obigem Beispiel ein Button vom Typ *editOrView* angegeben, wodurch in Abhängigkeit von den Berechtigungen entweder eine Bearbeiten-Schaltfläche oder eine Ansicht-Schaltfläche angezeigt wird.

In beiden Fällen wird beim Klicken auf diese Schaltfläche eine frontendseitige Framework-Funktion zum Aufruf eines Editors aufgerufen, der den selektierten Datensatz in Abhängigkeit von den vorliegenden Rechten im Bearbeiten- oder im Nur-Lesen-Modus anzeigt.

Die Definition der Schaltfläche über Button-Spalten bietet somit die Flexibilität, festzulegen, was im Falle fehlender Berechtigungen geschehen soll:

- Die betreffende Schaltfläche verstecken
- Die betreffende Schaltfläche deaktiviert (grau) darstellen
- Eine alternative Schaltfläche für eine reduzierte Funktion anzeigen (z. B. Editor im Readonly-Modus aufrufen)

Für das Editieren muss der zu verwendende Editor angegeben werden können. Dies erfolgt in obigem Beispiel über das Attribut *Values* der Abfrage-Definition, das eine Liste von Key/Value-Paaren enthält, wo u. a. eine Angabe für den Editor vorgenommen werden kann. Diese Liste wird von der o. g. frontendseitigen Framework-Funktion zum Aufruf des Editors ausgewertet.

```
72 , "Orders":
73   [ "RENTAL.RENTAL_DAY DESC"
74   , "MEMBER.LAST_NAME"
75   ]
76 , "Values":
77   [ { "key":"Editor", "value":"RENTAL_EDIT.htm"}
78   ]
79 }
```

Definitionsbeispiel 6: Angabe des Editors innerhalb einer Abfrage-Definition

4.1.6 Neuanlage

Die selbe Angabe wird auch für den Aufruf des Editors bei der Neu-Anlage über die Neu-Schaltfläche verwendet.

Weitere Definitionsmöglichkeit sind für die Neuanlage nicht erforderlich. Das Framework zeigt die Schaltfläche zur Neuanlage an, wenn die Berechtigung vorliegt – ansonsten nicht.

4.1.7 Anforderungen an das Framework

Zur Festlegung von Bearbeitungsschaltflächen sind folgende Definitionsmöglichkeiten und Funktionalitäten erforderlich:

1. Festlegung der Bearbeitungsmöglichkeiten pro Menüpunkt
2. Backendseitige Ausführung entsprechender Endpunkte nur bei ausreichender Berechtigung
3. Festlegung des zu verwendenden Editors
4. Frontendseitige Funktionen zum Aufruf von Editoren und zur Abfrage von Berechtigungen
5. Angabe der Abfrage-Spalten, die in einer Sicherheitsabfrage beim Löschen angezeigt werden sollen

4.2 Editoren

4.2.1 Zweck

Um dem Benutzer frontendseitig eine Eingabemöglichkeit zu bieten, die backendseitig zur Generierung eines schreibenden Statements führt, bedarf es einer Eingabemaske, hier Editor genannt.

4.2.2 Aufbau

Eine Eingabemaske besteht aus Eingabefeldern und anderen Bedienelementen, wie z. B. einer Schaltfläche zum Speichern des eingegebenen Datensatzes (siehe *Screenshot 2*, wo die beiden Editoren zur Bearbeitung einer *Ausleihe* und eines *Benutzers* gezeigt werden).

4.2.3 Auswertung

Editoren erfordern eine sehr individuelle Gestaltung. Hier bietet HTML (in Verbindung mit Bibliotheken wie z. B. jQuery[10]) umfangreiche Möglichkeiten und wird zudem während der Laufzeit vom Browser interpretiert, so dass eine HTML-Seite vergleichbar deskriptiv und separat auslieferbar ist wie eine Abfrage-Definitionsdatei. Daher sieht das hier beschriebene Framework keine zusätzlichen Definitionen für die Gestaltung von Editoren vor.

Der HTML-Code für den in *Screenshot 2* abgebildeten Editor für Ausleihen sieht z. B. wie folgt aus:

[10] jQuery ist eine JavaScript-Bibliothek, mit der eine Web-Anwendung auf einfache Weise auf ihre HTML-Elemente zugreifen und diese manipulieren kann [29].

```
1  <div id="htmlPageContent" udq-options="dialogClass:udq-dialog; width:1000px; height:500px">
2    <form>
3      <div id="CODE_lgMain" style="width:100%;">
4        <h2 id="txtLabel" class="headerLabel mlLabel">Ausleihe
5          <span class="PK-label"> | ID: </span>
6          <span id="dfnID" class="Bind-Number"></span>
7        </h2>
8
9        <div id="Grunddaten" class="ui-grid-a">
10         <div class="form-group breakpoint-input ui-block-a" style="width:10rem;">
11           <input type="text" id="dfdRENTAL_DAY" class="form-control dfDate Bind-Date" required>
12           <label class="form-control-placeholder mlLabel" for="dfdRENTAL_DAY">Ausgabedatum</label>
13         </div>
14
15         <div class="form-group breakpoint-input ui-block-b" style="width:10rem;">
16           <input type="text" id="dfdRETURN_DAY" class="form-control dfDate Bind-Date"
17            udq-lower-date="dfdRENTAL_DAY">
18           <label class="form-control-placeholder mlLabel" for="dfdRETURN_DAY">Rückgabedatum</label>
19         </div>
20
21         <div class="breakpoint-area ui-block-a">
22           <div id="sdtMEMBER_ID" class="sdt Bind-Number" udq-query="MEMBER_SDT" required></div>
23         </div>
24
25         <div class="breakpoint-area ui-block-b">
26           <div id="sdtBOOK_ID" class="sdt Bind-Number" udq-query="BOOK_SDT" required></div>
27         </div>
28       </div>
29     </div>
30   </form>
31 </div>
```

Definitionsbeispiel 7: Editor RENTAL_EDIT.htm für Ausleihe-Dialog

Hier wird u. a. von jQuery-Features wie Grids[11] Gebrauch gemacht, was jedoch im Rahmen dieser Arbeit nicht im Fokus der Betrachtung liegt.

Charakteristisch für das Framework ist hingegen, dass Bedienelemente über Klassen mit dem Namen Bind-Number, Bind-String oder Bind-Date verfügen. Hierüber werden sie als solche gekennzeichnet, die an eine Abfrage-Definition gebunden sind, die die Daten zwischen Editor und Backend überträgt. So gekennzeichnete Bedienelemente erhalten also ihren Wert aus der Abfrage des Editors und ihre Eingaben werden beim Speichern mittels der selben Abfrage-Definition in das entsprechende Datenbank-Statement eingefügt.

4.2.4 Datenselektierenden GUI-Komponenten des Editors

Des Weiteren verfügt der Editor neben elementaren Eingabefeldern auch über solche, die eigene Datenabfragen erfordern – wie beispielsweise die beiden SDTs für die beiden Verweise auf Benutzer (Zeile 22) und Buch (Zeile 26).

[11] Grids sind tabellarische Anordnungen von HTML-Elementen.

Diese datenselektierenden GUI-Komponenten verfügen in ihrer HTML-Definition jeweils über ein Attribut namens udq-query, in dem die ID des Menüpunktes für die Abfrage-Definition angegeben ist. Diese Menüpunkte sind nicht direkt über das Menü aufrufbar, das dem Anwender in der GUI angezeigt wird, sind jedoch Bestandteil des Menüs – sprich: des Datenangebots, das dem Anwender zur Verfügung steht, was gleichbedeutend mit seinem Rechteprofil ist.

Solche nicht direkt aufrufbaren Menüeinträge können in der Menü-Definition der Rolle des Benutzers unterhalb eines versteckten Menüpunktes untergebracht werden. Ein solcher versteckter Menüpunkt ist in den Zeilen 32 – 34 von *Definitionsbeispiel 1* zu finden. Dort wird eine ausgelagerte Menü-Definitionsdatei inkludiert, die Menüeinträge für datenselektierende GUI-Elemente enthält, die nicht direkt aus dem Menü aufgerufen werden können, sondern in Editoren verwendet werden.

```
1 [ { "Id":"LOCATION_CMB", "Type":"dropdown", "Label":"Standort", "File":"LOCATION.query"}
2 , { "Id":"USERROLE_CMB", "Type":"dropdown", "Label":"Rolle",    "File":"USERROLE.query"}
3 , { "Id":"MEMBER_SDT",   "Type":"sdt",      "Label":"Benutzer", "File":"MEMBER.query",  "CRUD":"CRUD"}
4 , { "Id":"BOOK_SDT",     "Type":"sdt",      "Label":"Bücher",   "File":"BOOK.query",    "CRUD":"CRUD"}
5 ]
```

Definitionsbeispiel 8: Datenselektierende GUI-Elemente für Editoren

In obigen Beispielen wird für den Editor stets die selbe Abfrage verwendet wie für den zugehörigen Browser. Es kann jedoch erforderlich sein, für den Editor mehr Attribute des bearbeiteten Objektes zu selektieren als im Browser angezeigt werden sollen. In solch einem Fall kann für den Editor ein separater versteckter Menüeintrag vom Type *editor* mit einer abweichenden Abfrage-Definition definiert werden.

Außerdem müssen Editoren ihre Daten autark abfragen können, um unabhängig von einem speziellen Browser aufgerufen werden zu können. Ein und der selbe Editor mag von verschiedenen Stellen aufgerufen werden müssen.

Dadurch, dass beispielsweise das im Ausleihe-Editor verwendete Benutzer-SDT im Attribut *CRUD* nicht nur Lese-, sondern auch Bearbeitungsrechte definiert hat (Zeile 3 aus obigem Ausschnitt der Menü-Definition), verfügt das Benutzer-SDT im Ausleihe-Editor über eine Bearbeiten-Schaltfläche, mittels derer ein Benutzer-Editor aufgerufen werden kann. Der selbe Editor kann auch über den Benutzer-Browser aufgerufen werden, der direkt aus dem Menü heraus erreichbar ist. SDT und Browser verwenden hier die selbe Abfrage-Definition *MEMBER.query*, die auch für den Benutzer-Editor ausreicht. Daher ist für diesen keine eigene Abfrage-Definition angegeben.

Dieses Beispiel zeigt zudem, dass ein und die selbe Abfrage-Definition für verschiedene Zwecke und in verschiedenen Darstellungsformen (und entsprechend unterschiedlichem Attribut *Type*) verwendet werden kann.

Da Abfrage-Definitionen in den vorangegangenen Kapiteln bereits ausführlich beschrieben wurden, wird an dieser Stelle auf den Inhalt der einzelnen Abfrage-Definitionen der datenselektierenden GUI-Komponenten für SDTs und Comboboxen nicht weiter eingegangen. Es sei lediglich erwähnt, dass diese Abfrage-Definitionen in ihrer Spalten-Liste mindestens eine Spalte mit einem Attribut *Constraint* und dem Wert *PK* besitzen, um kenntlich zu machen, welche Spalte(n) als Primärschlüssel verwendet wird/werden.

Der Primärschlüssel des ausgewählten Buches wird beispielsweise für den entsprechenden Fremdschlüssel in der Ausleihe benötigt.

4.2.5 Abfrage-Definition des Editors

Auch die Abfrage-Definition des Editors selbst muss einen Constraint *PK* haben, damit die zu bearbeitende Zeile eindeutig selektiert werden kann.

Die Abfrage-Definition für Ausleihen (*Definitionsbeispiel 2: Abfrage-Definition RENTAL.query*) enthält beispielsweise in ihrer Spalten-Liste eine Spalten-Definition, in der die Datenbankspalte *RENTAL.ID* als Primärschlüssel gekennzeichnet wird (Zeile 32). Des Weiteren wird dort angegeben, wie bei der Neuanlage einer Ausleihe ein neuer Wert für diese Spalte ermittelt werden soll. In diesem Falle wird *AUTO* angegeben, da die ID automatisch durch die Datenbank vergeben werden soll.

Des Weiteren enthält die Abfrage-Definition des Editors für jedes darzustellende Attribut des im Editor bearbeiteten Objekts je eine Spalten-Definition, die jeweils mit dem entsprechenden Bedienelement des Editors über die o. g. Bind-Klassen verknüpft ist.

Die mit Bind-Klassen gekennzeichneten HTML-Elemente sind über eine Namenskonvention an die Spalten-Definition der Abfrage-Definition geknüpft, wobei die id des HTML-Elements aus einem dreistelligen Präfix und dem jeweiligen Spaltennamen besteht – sprich: für das HTML-Element mit der id *dfdRENTAL_DAY* gibt es eine Spalten-Definition, die die Spalte *RENTAL_DAY* selektiert; des gleichen für *sdtMEMBER_ID*, *sdtBOOK_ID* u. s. w. (vergleiche *Definitionsbeispiel 2: Abfrage-Definition RENTAL.query* und *Definitionsbeispiel 7: Editor RENTAL*_EDIT.htm für Ausleihe).

4.2.6 Lebenszyklus eines Editors

Ein Editor durchläuft mehrere Phasen, in denen Abfragen ausgeführt werden müssen.

4.2.6.1 Laden der Seite und des bearbeiteten Datensatzes

Nach dem Laden der HTML-Seite für den Editor wird zunächst, sofern der Editor im Bearbeitungs- oder Nur-Lesen-Modus geöffnet wird, der zu bearbeitende Datensatz an Hand der Abfrage-Definition des Editors geladen.

Hierzu erhält der Editor den Primärschlüssel des zu bearbeitenden Datensatzes vom Aufrufer, bei dem es sich beispielsweise um einen Browser oder ein SDTs handeln kann – z. B. um den Ausleihe-Browser, aus dem der Ausleihe-Editor aufgerufen werden kann, oder um das im Ausleihe-Editor befindliche Benutzer-SDT, aus dem ein Benutzer-Editor aufgerufen werden kann.

Diesen PK gibt der Editor mit dem Request nach dem Datensatz an das Backend und erhält hierfür die Daten des zu bearbeitenden Datensatzes.

Mit diesen Daten werden die Bedienelemente gefüllt, die über ihre Bind-Klassen an die Spalten-Definitionen der Abfrage-Definition des Editors gebunden sind. Elementare Bedienelemente wie Eingabefelder können ihren so erhaltenen Wert direkt ausgeben.

Wird der Editor im Neuanlage-Modus geöffnet, wird natürlich kein Datensatz gelesen und die Bedienelemente werden geleert.

4.2.6.2 Laden der Daten datenselektierender Komponenten

In der darauffolgenden Phase wird für jede datenselektierende Komponenten wie SDTs, Comboboxen oder auch Master-Detail-Tabellen jeweils die Abfrage-Definition der jeweiligen Komponente ausgeführt, damit diese Komponenten ihre Daten erhalten und darstellen können.

SDTs innerhalb von Editoren werden dafür genutzt, um Detail-Daten zu einem Objekt anzuzeigen, auf das der vom Editor bearbeitete Datensatz per Fremdschlüssel referenziert. Für den Wert dieses Fremdschlüssels wird also die Abfrage-Definition des SDT ausgeführt, sofern der Datensatz des Editors für diesen Fremdschlüssel einen Wert hat. Ansonsten wird das SDT im Suchkriterien-Modus angezeigt, was z. B. im Neuanlage-Modus des Editors regelmäßig der Fall ist. In obigem Beispiel des Ausleihe-Editors besitzt der Datensatz der Ausleihe einen Fremdschlüssel auf den Benutzer, der sich in der Tabelle *MEMBER* befindet. Das SDT zeigt demnach *MEMBER*-Daten des von der Ausleihe referenzierten Benutzers an.

Auch Comboboxen sind i. d. R. mit einem Fremdschlüssel verknüpft – und zwar mit einem Fremdschlüssel, mit dem der Datensatz des Editors auf einen Katalog-Eintrag verweist. In obigem Beispiel des Benutzer-Editors sind je eine Combobox für die Rolle und für den Standort enthalten. In beiden Fällen listet die Combobox

die Daten eines Katalogs (Auswahl der möglichen Rollen und Auswahl von Standorten) auf. Der Benutzer-Datensatz verweist mit einem Fremdschlüssel auf einen Katalogeintrag. Die Combobox muss den gesamten Katalog auflisten und den Eintrag, auf den das editierte Objekt (hier der Benutzer) verweist, beim Laden des Datensatzes selektieren, sofern der betreffende Fremdschlüssel im bearbeiteten Datensatz gefüllt ist, was z. B. im Neuanlage-Modus nicht der Fall ist.

Bei Tabellen aus einer Master-Detail-Tabelle ist die 1:n-Beziehung umgekehrt. Hier wird der bearbeitete Datensatz über einen Fremdschlüssel aus den Daten der Detail-Tabelle referenziert. Bzgl. Details wird auf den Abschnitt *Darstellung von untergeordneten Objekten* verwiesen. Im Neuanlage-Modus bleiben Master-Detail-Tabellen leer.

4.2.6.3 Speichern

Sollen die Eingaben gespeichert werden, sendet der Client einen entsprechenden Request an den Server und liefert die Daten der über Bind-Klassen gebundenen Bedienelemente mit.

Je nachdem, ob der Editor im Bearbeiten- oder Neuanlage-Modus geöffnet wurde, sendet der Client entweder einen Update- oder einen Insert-Request an den Server. Hierzu verfügt der Server über entsprechende Endpunkte – und zwar über genau einen für Update und einen für Insert – unabhängig davon, aus welchem Editor heraus der Aufruf erfolgen kann. Alle Editoren verwenden die selben beiden Endpunkte und geben neben ihren Daten die ID ihres Menüpunktes mit, so dass die Endpunkte an Hand dieser ID die jeweilige Abfrage-Definition ermitteln und daraus das jeweilige Update- oder Insert-Statement generieren können.

Das Update-Statement, das das Backend des Frameworks für den Ausleihe-Editor aus dessen Abfrage-Definition generiert, sieht wie folgt aus:

```
1 UPDATE RENTAL
2    SET RENTAL_DAY  = :Item_0
3      , RETURN_DAY  = :Item_1
4      , MEMBER_ID   = 94002
5      , BOOK_ID     = 3039
6  WHERE ( RENTAL.ID = 168416 )
```

Statement 3: Aus Abfrage-Definition generiertes Update-Statement

Zur Generierung des UPDATE- oder auch INSERT-Statements geht das Framework die Spaltenliste der Abfrage-Definition durch und ermittelt die Spalte(n), bei der/denen das PK-Constraint gesetzt ist. Für die Tabelle der ersten so ermittelten Spalte wird das UPDATE- bzw. INSERT-Statement generiert, indem alle Spalten mit dem selben *Table*-Attribut in die Spalten- und Werteliste des Statements aufgenommen werden. PK-Spalten werden dabei in die WHERE-Bedingung

aufgenommen, wenn es sich um ein UPDATE-Statement handelt. Bei INSERT-Statements werden PK-Spalten ebenfalls in die Spalten-Liste aufgenommen und es wird pro PK-Spalte ein PK-Wert generiert und in die Werte-Liste aufgenommen, sofern der PK nicht vom Datenbankserver generiert werden soll.

Zur Vermeidung von SQL-Injection[12] werden die Werte für Zeichenketten-Spalten in Bind-Variablen übertragen und diese Bind-Variablen werden ins Statement aufgenommen. Auch Datumswerte werden über Bind-Variablen eingefügt, um nicht für die jeweilige Datumseinstellung des Datenbankservers passende Datumsliterale formatieren zu müssen. Numerische Werte können dagegen als Literal ins Statement eingefügt werden, wenn sie zuvor entsprechend plausibilisiert wurden.

Ferner muss das Backend prüfen, ob die jeweilige Operation überhaupt im Rechteprofil der Rolle des Anwenders erlaubt wurde, damit unbefugte Update- oder Insert-Requests abgewiesen werden können.

4.2.6.4 Datensatz neu laden

Nach einem erfolgreichen Speichervorgang muss der Datensatz erneut geladen werden, sofern der Editor nicht sofort nach dem Speichern geschlossen werden soll.

Es kann nämlich sein, dass durch den Speichervorgang zusätzlich zu den Daten der bearbeiteten Bedienelemente auch weitere Daten des Datensatzes oder von ihm referenzierte Daten aktualisiert wurden.

Dies ist etwa dann der Fall, wenn es nicht editierbare Bedienelemente gibt, die durch backendseitige Programmlogik in Abhängigkeit von editierten Bedienelementen verändert werden oder bei automatisch generierten Werten, wie z. B. der Benutzer-Nr., die generiert wird und im Benutzer-Editor nur angezeigt wird.

Solche backendseitige Programmlogik kann beispielsweise in Datenbank-Triggern erfolgen oder aber in backendseitigen Extension-Points, auf die später im Text näher eingegangen wird (siehe *Backendseitige Extension-Points*).

---

[12] Mit SQL-Injection werden Hacker-Angriffe bezeichnet, bei denen die von der Anwendung ausgeführten Datenbank-Statements manipuliert werden. Wenn beispielsweise Eingaben direkt in den Aufbau eines serverseitig ausgeführten Statements einfließen, kann durch geschicktes Setzen dieser Eingaben ein Statement um Bedingungen ergänzt werden, die programmseitige Bedingungen außerkraftsetzen.

Es kann mehrere Gründe geben, den Editor nicht sofort nach dem Speichern zu schließen. Beispielsweise kann aus einem Editor heraus ein Report aufgerufen werden, was ein Speichern der Eingaben erfordert, damit der Report, der die Daten aus der Datenbank beziehen mag, die eingegebenen Daten berücksichtigen kann. Dieses Speichern erfolgt implizit, ohne dass der Benutzer explizit auf eine Schaltfläche o. ä. zum Speichern klickt und den Editor zu schließen.

Das erneute Laden nach dem Speichern mit Verbleib im Editor betrifft auch die datenselektierenden GUI-Komponenten.

4.2.6.5 Aufrufer aktualisieren

Weiterhin muss auch der Aufrufer des Editors aktualisiert werden. Hierbei kann es sich um einen Browser handeln (z. B. den Ausleihe-Browser, aus dem der Ausleihe-Editor aufgerufen wurde) oder um ein SDT (z. B. ein Benutzer-SDT, aus dem ein Benutzer-Editor aufgerufen wurde).

4.2.7 Anforderungen an das Framework

Das Framework muss folgendes unterstützen:

1. Aufruf von Editoren im Neuanlage-Modus
2. Editor-Aufruf im Bearbeiten-Modus und Mitgabe des Primärschlüssels
3. Selektion an Hand der Abfrage-Definition des Editors unter Berücksichtigung des Primärschlüssels
4. Bindung von Bedienelementen an Spalten-Definitionen der Abfrage-Definition
5. Durchlaufen der o. g. Phasen des Lebenszyklus eines Editors inkl. der pro Phase erforderlichen Selektionen und Datentransporte zwischen Client und Server
6. Generieren von UPDATE- und INSERT-Statements an Hand der Abfrage-Definition des Editors
7. Generierung von Primärschlüsseln
8. Dynamische Erzeugung von Bind-Variablen
9. Validierung des Datentyps bei numerischen Wert-Literalen
10. Backendseitige Überprüfung der Berechtigung für die jeweiligen Operationen an Hand der Menü-Definition

4.3 Rollen-Filter bei schreibenden Zugriffen

4.3.1 Zweck

Wie bereits im Kapitel *Filter-Definitionen* beschrieben, dienen solche Filter dem Zweck, Anwendern nur eine eingeschränkte Sicht auf definierte Zeilen einer Tabelle zu gewähren.

Darf ein Anwender auf Grund seiner Rolle nur bestimmte Daten selektieren (z. B. nicht alle Ausleihen, sondern nur Ausleihen der Benutzer seines Standorts), dann soll er natürlich auch nur diese editieren dürfen. Daher spielen die oben beschriebenen Filter-Definitionen auch bei schreibenden Zugriffen eine erhebliche Rolle.

4.3.2 Auswertung

Speichervorgänge müssen also backendseitig dahingehend abgesichert werden, dass sie sich innerhalb des Sichtbarkeitsbereichs des Benutzers abspielen.

Sendet ein Client ein UPDATE-Request für einen bestimmten Primärschlüssel, muss der betroffene Datensatz backendseitig erneut an Hand dieses Primärschlüssels selektiert werden. Dieser Selektionsvorgang muss exakt einen Datensatz finden, da im Editor immer nur ein Datensatz bearbeitet wird – der des übergebenen Primärschlüssels. Befindet sich der über den Primärschlüssel selektierte Datensatz außerhalb des durch die Rollen-Filter-Definitionen geregelten Sichtbarkeitsbereichs des Anwenders, wird kein Datensatz gefunden und der UPDATE-Request wird abgewiesen.

Ähnliches gilt für einen DELETE-Request. Ein zu löschender Datensatz muss sich im Sichtbarkeitsbereich des Benutzers befinden, weshalb vor dem Absetzen des DELETE-Statements eine solche Kontrollabfrage stattfinden muss.

Ebenso muss auch nach einem UPDATE und auch nach einem INSERT eine solche Kontroll-Selektion erfolgen, um sicherzustellen, dass der Datensatz nicht so verändert wurde, dass er aus dem Sichtbarkeitsbereich des Benutzers herausfällt. Fiele er heraus, müsste die Transaktion zurückgerollt werden.

Dies kann beispielsweise dadurch passieren, dass ein UPDATE-Request in den mitgegebenen Eingabedaten ungültige Fremdschlüssel enthält. Der Request samt Eingabedaten muss schließlich nicht zwingend aus der oben beschriebenen GUI des Editors stammen, sondern kann auch unabhängig davon abgesetzt oder manipuliert worden sein. Letzteres ist in einer Web-Anwendung sehr leicht durchzuführen: Man fängt den Request ab, den die GUI beim Speichern absetzt,

ändert die mitgegebenen Eingabedaten und leitet ihn erst dann an den Server weiter.

Der oben beschriebene Ausleihe-Editor enthält beispielsweise über das Benutzer-SDT die Eingabe- bzw. Auswahlmöglichkeit des Benutzers der Ausleihe (hier ist mit „Benutzer" ein Benutzer der Bibliothek, also derjenige gemeint, der sich das im Ausleihe-Datensatz angegebene Buch leiht). Die GUI präsentiert bei der Auswahl von Benutzern nur solche, die über die Abfrage-Definition des SDT selektiert werden – also nur solche, die dem Rollenfilter für Benutzer entsprechen. Wenn der Anwender der Anwendung (man beachte die in diesem Dokument vorgenommene Unterscheidung zwischen den Begriffen „Benutzer" und „Anwender") nur für einen Standort berechtigt ist, bekommt er also nur Benutzer seines Standorts als mögliche Benutzer der Ausleihe angeboten.

Sei beispielsweise der Benutzer 4711 innerhalb des Sichtbarkeitsbereichs des angemeldeten Anwenders, der Benutzer 4710 hingegen nicht, könnte durch Austausch der *MEMBER_ID* im UPDATE-Request der Ausleihe von 4711 auf 4710 ein UPDATE veranlasst werden, durch den die Ausleihe zu einem Benutzer eines anderen Standorts gehört und dadurch aus den Sichtbarkeitsbereich des Anwenders heraus fällt. Manipulationen dieser Art sind insbesondere dann leicht realisierbar, wenn als Schlüssel wie in den oben vorgestellten Beispielen fortlaufende Nummern an Stelle von GUIDs[13] verwendet werden. Das Vorhandensein eines Benutzers mit der ID 4710 ist schließlich nicht schwer zu erraten, wenn es im Sichtbarkeitsbereich des Anwenders einen Benutzer mit der ID 4711 gibt.

Oben beschriebene Manipulation scheint auf den ersten Blick den Interessen eines manipulierenden Anwenders zu widersprechen, weil er dadurch keinen Einblick in Daten erhält, zu denen er nicht berechtigt ist, sondern im Gegenteil Daten aus seinem Sichtbarkeitsbereich herausfallen.

Doch zum einen ist das vermeintliche Fehlen eines Manipulationsmotivs kein ausreichender Sicherheitsmechanismus. Und zum anderen kennt man als Entwicklerin eines Frameworks nicht die letztendlichen Einsatzgebiete der damit entwickelten Anwendungen – ob es sich beispielsweise um eine nur firmenintern oder gar nur durch einen Anwender verwendete Anwendung handelt oder ob sie im Mehrmandantenbetrieb über das Internet jedermann zur Verfügung steht. Wenn beispielsweise, wie im Falle der Fuhrparkmanagementsoftware Fleet+

[13] GUID steht für *globally unique identifier* und bezeichnet eine 16 Byte lange Hexadezimalzahl, die durch einen Algorithmus mit geringer Kollisionswahrscheinlichkeit generiert und daher als ID verwendet werden kann.

Web, die Nutzungsgebühren von der Anzahl der verwalteten Fahrzeuge abhängt, könnte man durch obige Manipulation unmittelbar vor dem Abrechnungsstichtag seine Nutzungsgebühren „kleinrechnen", indem man seine Fahrzeuge datentechnisch einem anderen Mandanten zuschiebt.

Eine solche Manipulation kann jedoch durch die Kontroll-Selektion im Anschluss an das Update abgefangen werden, so dass die Transaktion dann abgebrochen werden kann.

4.3.3 Aufbau

Dies setzt allerdings voraus, dass die Abfrage-Definition eines Editors in ihrer Tabellenliste auch die Tabellen enthält, auf die sich sichtbarkeitsrelevante editierbare Fremdschlüssel beziehen.

In oben beschriebenem Fall enthält die Tabellenliste der Abfrage-Definition des Ausleihe-Editors zusätzlich zur Tabelle *RENTAL* auch einen JOIN zur Tabelle *MEMBER*. Damit würden Rollenfilter auf die Tabelle *MEMBER* greifen, die bei einem im Request enthaltenen Benutzer aus einem nicht berechtigten Standort dazu führen, dass die Ausleihe mit dem Kontrollstatement nicht mehr gefunden würde.

Im Falle von Fremdschlüsseln aus per OUTER JOIN verknüpften Tabellen ist zusätzlich eine Filter-Bedingung in der Filter-Liste der Abfrage-Definition des Editors anzugeben, die sicherstellt, dass der Fremdschlüssel im UPDATE-Statement entweder nicht gesetzt wurde oder so, dass der Eintrag auch im Sichtbarkeitsbereich des Anwenders gefunden werden kann.

4.3.4 Anforderungen an das Framework

Damit die durch Filter-Definitionen geregelten Sichtbarkeitsgrenzen auch bei schreibenden Zugriffen gewahrt bleiben, muss das Framework bei Schreibvorgängen folgendes tun:

1. Vor dem Ausführen von UPDATE- und DELETE-Statements den betroffenen Datensatz selektieren und die Aktion abweisen, wenn nicht genau ein Datensatz gefunden wird.
2. Nach dem Ausführen von UPDATE- und INSERT-Statements den Datensatz selektieren und die Transaktion zurückrollen, wenn nicht genau ein Datensatz gefunden wird.

4.4 Referentielle Integrität beim Löschen

4.4.1 Zweck

Referentielle Integrität dient der Konsistenz der Verknüpfungen zwischen den Zeilen unterschiedlicher Datenbanktabellen.

So darf beispielsweise die Spalte *RENTAL.MEMBER_ID*, die den Benutzer einer Ausleihe enthält, nicht auf einen nicht in der Tabelle *MEMBER* existierenden Benutzer verweisen.

Wenn bei der Anlage der Datenbanktabellen darauf geachtet wird, dass Beziehungen mit entsprechenden Fremdschlüssel-Constraints versehen sind, stellt das Datenbankmanagementsystem die Referentielle Integrität sicher, so dass keine Zeilen mit ungültigen Fremdschlüsseln angelegt werden können und es auch nicht möglich ist, eine Zeile durch ein UPDATE so zu verändern, dass sie einen ungültigen Fremdschlüsselwert erhält. Auch beim Löschen stellt das Datenbankmanagementsystem sicher, dass keine Zeile gelöscht werden kann, die von einer anderen über einen Fremdschlüssel referenziert wird.

Dennoch ist es anwendungsseitig wünschenswert, nicht mögliche Lösch-Versuche gar nicht erst abzusetzen und dem Anwender eine aussagekräftige Fehlermeldung zu präsentieren.

Außerdem gibt es aus fachlicher Sicht unterschiedliche Szenarien, die zu beachten sind:

1. Löschvorgänge, die unterbunden werden sollen, wenn die zu löschende Zeile von anderen referenziert wird
2. Löschvorgänge, die referenzierende Zeilen von der zu löschenden Zeile lösen
3. Löschvorgänge, die das zusätzliche Löschen referenzierender Zeilen einschließen sollen (sog. cascading delete)
4. Löschvorgänge, die das zusätzliche Löschen referenzierter Zeilen einschließen sollen, damit keine verwaisten Zeilen übrig bleiben
5. Löschvorgänge mit gegenseitigen Abhängigkeiten

In einem Framework, das eine Lösch-Schaltfläche durch die bloße Lösch-Berechtigung zur Verfügung stellt (siehe Abschnitt *Schaltflächen*), ist es erforderlich, obige Fälle ebenfalls deklaratorisch abzubilden.

4.4.2 Aufbau

Hierzu verfügt das Framework über eine zentrale Stelle, wo der Umgang mit Beziehungen bei Löschungen festgelegt werden kann. Dies erfolgt, wie bei den übrigen Definitionen auch, ebenfalls über eine JSON-Datei:

```
{ "BOOK":
  { "PrimaryKey":"ID"
  , "prevent":  // Tabellen, die von BOOK abhängig sind
    [ { "Table":"RENTAL", "ForeignKey":"ID"
      , "Condition":"RENTAL.BOOK_ID = #<id># AND RENTAL.RETURN_DAY IS NULL"
      , "Message":"Bücher mit offenen Ausleihen können nicht gelöscht werden"
      }
    ]
  , "delete":  // cascading delete
    [ { "Table":"RENTAL", "ForeignKey":"ID"
      , "Condition":"RENTAL.BOOK_ID = #<id># AND RENTAL.RETURN_DAY IS NOT NULL"
      }
    ]
  }
, "MEMBER":
  { "PrimaryKey":"ID"
  , "prevent":  // Tabellen, die von MEMBER abhängig sind
    [ { "Table":"RENTAL", "ForeignKey":"ID"
      , "Condition":"RENTAL.MEMBER_ID = #<id># AND RENTAL.RETURN_DAY IS NULL"
      , "Message":"Benutzer mit offenen Ausleihen können nicht gelöscht werden"
      }
    ]
  , "delete":  // cascading delete
    [ { "Table":"RENTAL", "ForeignKey":"ID"
      , "Condition":"RENTAL.MEMBER_ID = #<id># AND RENTAL.RETURN_DAY IS NOT NULL"
      }
    ]
  }
}
```

Definitionsbeispiel 9: Löschregeln check_delete.json

Die Datei hat die Struktur vom Typ Dictionary mit je einem Attribut pro Tabelle, für die die Entwicklerin eine Festlegung treffen möchte, was bei Löschungen in der jeweiligen Tabelle geschehen soll.

Der Wert für jede Tabelle ist wiederum ein Dictionary mit den Attributen

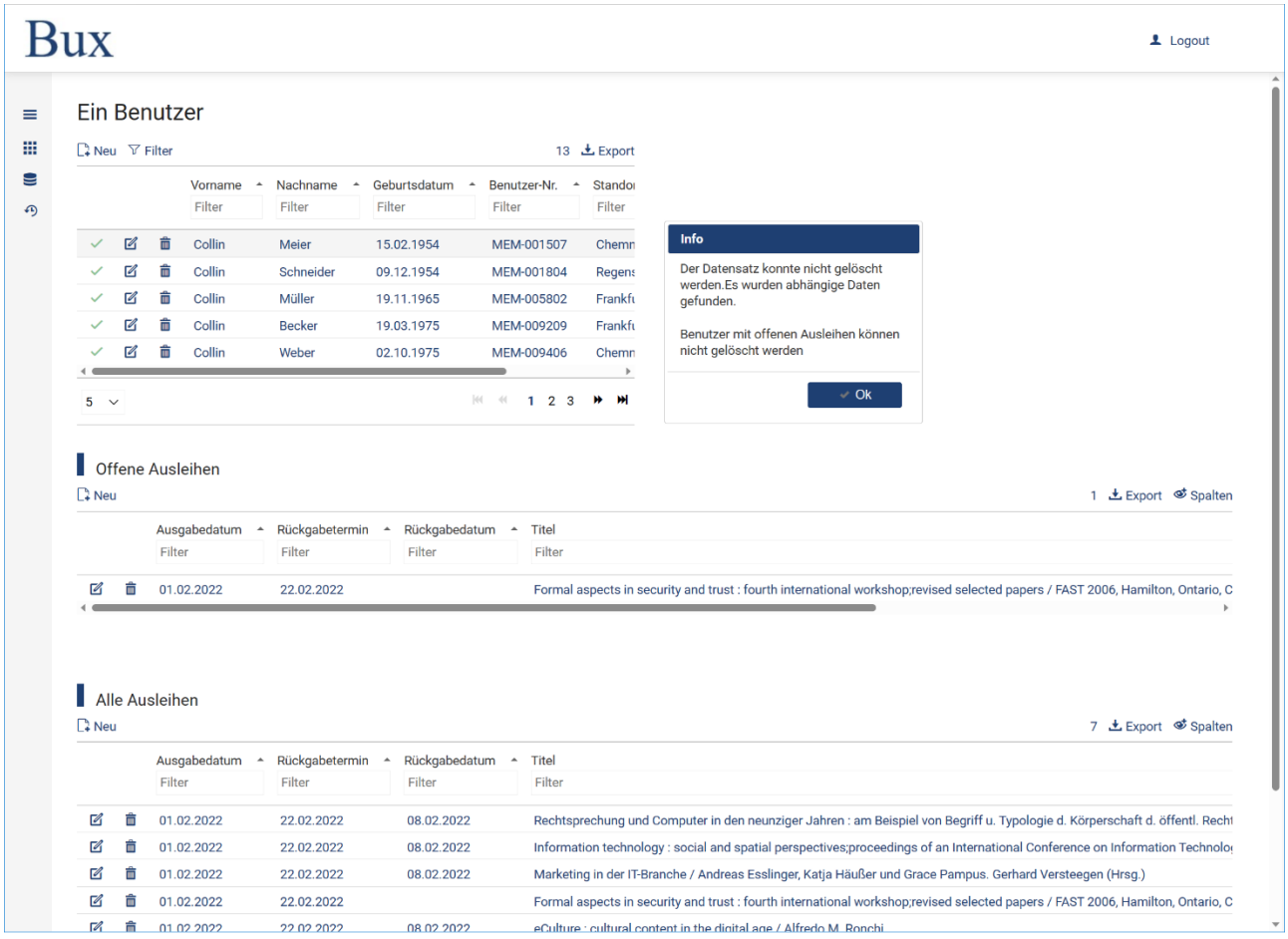
1. *prevent* mit einer Liste von Tabellen, die das Löschen verhindern, wenn in ihnen ein Verweis auf die zu löschende Zeile existiert (Szenario 1 aus der Einleitung dieses Kapitels),
2. *release* mit einer Liste von Tabellen, deren Bezüge auf die zu löschende Zeile aufgehoben werden sollen (Szenario 2) und
3. *delete* mit einer Liste von Tabellen, deren Zeilen, die auf die zu löschende Zeile verweisen, ebenfalls gelöscht werden sollen (Szenarien 3 bis 5).

Jeder Listeneintrag ist jeweils ein Dictionary mit zumindest dem Attribut *Table*.

4.4.3 Auswertung

4.4.3.1 prevent

Für die Tabellen in der prevent-Liste wird vor der Löschung jeweils selektiert, wie viele Zeilen auf die zu löschende Zeile referenzieren und dem Anwender dann eine entsprechende Meldung gegeben, wenn auf Grund solcher Bezüge nicht gelöscht werden kann (Szenario 1).



Screenshot 6: Meldung bei verhinderter Löschung

Aus obiger prevent-Liste würde somit nachstehendes Statement generiert werden.

```
SELECT ( SELECT COUNT(*) AS ANZAHL
           FROM RENTAL
          WHERE RENTAL.MEMBER_ID = :PK AND RENTAL.RETURN_DAY IS NULL
       ) AS COUNT_TABLE_RENTAL
  FROM MEMBER
 WHERE MEMBER.ID = :PK
```

Statement 4: Kontroll-Statement für die prevent-Liste

Der Spaltenname der Primärschlüsselspalte in der zu löschenden Tabelle und der Spaltenname der Fremdschlüsselspalte in den referenzierenden Tabellen werden per Namenskonvention (Tabellen-Name zzgl. _ID) abgeleitet, sofern dies nicht explizit abweichend in den Attributen *PrimaryKey* für die zu löschende bzw. *ForeignKey* für die referenzierende Tabelle definiert ist.

Wenn eine Zeile in einer referenzierenden Tabelle grundsätzlich das Löschen in der referenzierten verhindern soll, reicht die Angabe der referenzierenden Tabelle in der prevent-Liste (und ggf. die Angabe des von der Namenskonvention abweichenden Fremdschlüssels).

Wenn die Verhinderung jedoch von Geschäftslogik abhängig ist, kann diese über eine Bedingung angegeben werden, wie dies in obigem Definitionsbeispiel bei den Ausleihen der Fall ist, die das Löschen von Büchern oder Benutzern nicht grundsätzlich verhindern, sondern nur, wenn noch keine Rückgabe erfolgt ist.

Datensätze der Tabelle *RENTAL* mit offener Rückgabe (`RENTAL.RETURN_DAY IS NULL`) verhindern die Löschung des von ihnen referenzierten Datensatzes in der Tabelle *BOOK* (Szenario 1). Das selbe gilt für den Versuch, einen Datensatz aus *MEMBER* zu löschen, für den es noch eine offene Ausleihe gibt. Ist in der Ausleihe jedoch die Rückgabe angegeben (`RENTAL.RETURN_DAY IS NOT NULL`), so werden referenzierende Datensätze der Tabelle *RENTAL* beim Löschen aus *BOOK* oder *MEMBER* mit gelöscht (Szenario 3).

Die Angabe einer Bedingung ist auch dann sinnvoll, wenn kein direkter Bezug besteht, sondern die zu löschende Zeile und die davon betroffenen „über mehrere Ecken“ miteinander verknüpft sind.

Die prevent-Liste muss nicht sämtliche von der referentiellen Integrität betroffenen Tabellen enthalten, sondern nur diejenigen, für die die Entwicklerin dem Anwender eine sprechende Fehlermeldung geben möchte, wenn nicht gelöscht werden kann. Dass tatsächlich nicht gelöscht werden kann, solange Abhängigkeiten bestehen, ist über entsprechende Constraints in der Datenbank sicherzustellen. Löschversuche bei bestehenden Abhängigkeiten führen zu entsprechenden Fremdschlüsselverletzungen auf Datenbankseite. Die Interpretation der SQL-Fehlermeldung und eine entsprechende Überführung in einen benutzerfreundlichen Hinweis ist nicht Gegenstand dieser Arbeit.

4.4.3.2 release

Zeilen in Tabellen aus der release-Liste werden vor dem Löschen von dem zu löschenden Datensatz gelöst (Szenario 2).

Beispielsweise könnte ein Katalog mit Genres definiert werden und jedem Buch optional ein Genre zugeordnet werden. Wenn ein Genre gelöscht wird, könnte gefordert sein, diese Zuordnung von allen Büchern zu entfernen, die diesem Genre bislang zugeordnet waren.

Auch hier werden die betreffenden Spaltennamen mittels Namenskonvention abgeleitet oder es wird eine Bedingung angegeben, mit der die entsprechenden Zeilen gefunden werden können.

4.4.3.3 delete

In ihrer Grundform (Szenario 3) enthält die delete-Liste Tabellen, die von einem cascading delete betroffen werden sollen, wobei auch hier wieder über Bedingungen indirekte Bezüge oder fachliche Regeln realisiert werden können.

Die delete-Liste kennt zudem zwei zusätzliche Optionen:

1. after (Szenario 4)
2. reset (Szenario 5)

Das Löschen referenzierter Zeilen, die ansonsten verwaist übrig blieben, wird über die Option *after* angegeben.

Beispielsweise könnte die Adresse jedes Benutzer der Tabelle *MEMBER* über eine mittels Fremdschlüssel verknüpfte Zeile der Tabelle *ADDRESS* angegeben sein.

Eine Adresse ist ohne den sie beziehenden Benutzer nutzlos und muss daher mit gelöscht werden, was allerdings auf Grund der durch das Datenbankmanagementsystem sichergestellten referentiellen Integrität erst nach der Löschung des Benutzers aus der Tabelle *MEMBER* erfolgen kann, da die *ADDRESS_ID* ein Fremdschlüssel in der Tabelle *MEMBER* ist.

Alle anderen cascading deletes erfolgen im Unterschied hierzu vor der die Kaskade auslösenden Löschung, da die kaskadierend zu löschenden abhängigen Datensätze einen Fremdschlüssel haben, der auf die zu löschende Zeile verweist.

Nun kann man darüber diskutieren, ob es sich bei der Fremdschlüsselbeziehung von *MEMBER* auf *ADDRESS* nicht um einen Design-Fehler handelt und ob es nicht sinnvoller wäre, in der Tabelle *ADDRESS* einen Fremdschlüssel auf die Tabelle *MEMBER* vorzusehen, damit jeder Benutzer mehrere Adressen haben kann, aber

es müssen auch die Fälle behandelt werden können, wo man auf ungünstige (historische) Design-Entscheidungen trifft, an denen man nicht immer etwas ändern kann.

Gleiches gilt für die delete-Option reset. Diese ist vornehmlich für wechselseitige Bezüge vorgesehen (Szenario 5). Dabei wird zunächst die wechselseitige Beziehung aufgelöst und dann gelöscht.

4.4.4 Anforderungen an das Framework

Das Framework muss die Möglichkeit bieten,

1. alle fünf oben beschriebenen Szenarien deklarativ angeben zu können,
2. hilfreiche Meldungen zu geben, sofern nicht gelöscht werden kann,
3. abweichende Meldungen per Definition angeben zu können,
4. eine sinnvolle Namenskonvention zu verwenden, um den Definitionsumfang knapp und prägnant zu halten,
5. von der Namenskonvention abweichende Angaben machen zu können und
6. Beziehungen als Bedingung angeben zu können

5 Wiederverwendung und Überladung

5.1 Wiederverwendung von Menü-Definitionen

5.1.1 Zweck

Das Framework arbeitet nach dem Prinzip „Konvention vor Konfiguration“, was in Dateinamenskonventionen zum Ausdruck kommt. So wird der Dateiname des Menüs einer Rolle aus dem Namen der Rolle abgeleitet. Des gleichen greift eine Filter-Definition beim Zugriff auf eine bestimmte Tabelle durch einen Anwender einer bestimmten Rolle, wenn die Filter-Definitionsdatei einen entsprechenden Namen hat.

Es kann jedoch gewünscht sein, ein und die selbe Menü-Definition für mehrere Rollen zu verwenden – insbesondere dann, wenn die Menü-Definition sehr umfangreich ist im Vergleich zu den Unterschieden zwischen den einzelnen Rollen.

5.1.2 Aufbau

Die Wiederverwendung einer Menü-Definition einer Rolle für eine andere Rolle kann dadurch realisiert werden, dass es für letztere eine Zuordnungsdatei gibt, die angibt, dass an Stelle der laut Namenskonvention erwarteten Datei eine Datei mit einem anderen Namen verwendet werden soll.

```
1 [ {"key": "MEMBER.menu", "value": "ADMIN.menu"}
2 ]
```

Definitionsbeispiel 10: Zuordnungsdatei MEMBER.map für Rolle MEMBER

5.1.3 Auswertung

Das Framework prüft zunächst, ob es für die laut Namenskonvention erwartete Datei eine abweichende Definition in der Zuordnungsdatei gibt und verwendet, sofern dies der Fall ist, die dort angegebene Datei.

In obigem Beispiel würde für die Rolle *MEMBER* die selbe Menü-Definitionsdatei verwendet werden wie für die Rolle *ADMIN*.

Da jedoch unterschiedliche Rollen i. d. R. auch unterschiedliche Berechtigungen haben, müssen diese Abweichungen auch bei Wiederverwendung einer Menü-Definition definiert werden können.

Hierzu dienen Delta-Dateien, die Abweichungen zur wiederverwendeten Menü-Definition angeben.

```
[ { "Id":"OVERVIEW_DASH",   "CRUD":"F"}
, { "Id":"MEMBER_DASH",     "Autostart":"yes", "Label":"Meine Daten"}
, { "Id":"MEMBER_DASH_SDT", "FilterCriteria":"no"}
, { "Id":"MEMBER",          "CRUD":"R"}
, { "Id":"BOOK",            "CRUD":"R"}
, { "Id":"RENTAL",          "CRUD":"R"}
]
```

Definitionsbeispiel 11: Delta-Datei MEMBER.delta für die Rolle MEMBER

Auf diese Weise kann einfach angegeben werden, wie das Rechteprofil ausgehend von einer wiederverwendeten Menü-Definition endgültig aussehen soll.

Bei der Bereitstellung des Menüs für den Anwender einer bestimmten Rolle wird also zunächst an Hand der Zuordnungsdatei geschaut, welche Menü-Definition tatsächlich verwendet werden soll, und anschließend werden Attribute der Menü-Elemente mit abweichenden Werten überladen, sofern in einer Delta-Datei solche Abweichungen festgelegt sind.

Abweichend definierte Menü-Elemente werden dabei an Hand ihrer ID identifiziert und pro ID kann jedes Attribut eines Menü-Elements mit einem abweichenden Wert versehen werden.

Obiges Beispiel zeigt, dass zunächst das Dashboard *Bibliothek* mit der gesamten Übersicht über den gesamten Bibliotheksbestand (siehe *Screenshot 1*) für Anwender der Rolle *MEMBER* ausgeblendet wird. Stattdessen wird der Menüpunkt mit der Übersicht zu einem Benutzer als Programmstart angegeben (Attribut *Autostart*) und dessen Label so überladen, dass dies der Perspektive eines Anwenders der Rolle *MEMBER* besser entspricht.

Da ein Anwender dieser Rolle sowieso nur seine Daten sehen darf, macht es keinen Sinn, wenn in dem führenden SDT dieses Dashboards Suchkriterien angeboten werden. Stattdessen kann sofort gesucht werden, was durch das Überladen des Attributs *FilterCriteria* geregelt wird.

Letztlich wird noch das Attribut *CRUD* der drei Menüpunkte, über die man zu den Editoren gelangt, auf „nur lesen“ gesetzt, so dass Anwendern dieser Rolle keine schreibenden Zugriffe möglich sind (siehe *Tabelle 2: Rollen der Beispielanwendung* und *Abbildung 4: Vertikale Differenzierung der Rollen*).

Ähnlich ist für die Rolle *EMPLOYEE* vorzugehen. Auch hier wird über eine Zuordnungsdatei angegeben, dass das Menü der Rolle *ADMIN* wiederzuverwenden ist. Die Delta-Datei der Rolle *EMPLOYEE* benötigt lediglich einen Eintrag, der die Berechtigung des Menüpunkts *BOOK* auf „nur lesen“ beschränkt.

Die oben beschriebene Auswertung – sprich: das Ermitteln, welche Menü-Definitionsdatei verwendet werden soll und welche Attribute abweichend zu setzen sind – sollte, da sie bei großen, tief verzweigten Menüs zeitaufwendig sein kann, nicht bei jedem Abruf eines Menüs, also nicht bei jedem Anmeldevorgang eines einzelnen Anwenders, erfolgen, sondern nur beim erstmaligen Abruf des Menüs einer jeden Rolle. Das dann für diese Rolle ermittelte Menü kann backendseitig gecached werden, so dass beim nächsten Anmeldevorgang eines Anwenders der selben Rolle auf das bereits ermittelte effektive Menü zurückgegriffen werden kann. Dieser Cache muss bei Änderung von Menü-, Zuordnungs- und Delta-Dateien zurückgesetzt werden, so dass Änderungen zur erneuten Ermittlung des effektiven Menüs führen.

5.1.4 Anforderungen an das Framework

Zur Wiederverwendung von Menü-Definitionen ist es erforderlich, definieren zu können, dass

1. von der Namenskonvention abweichende Dateien herangezogen werden sollen, und dass
2. die Werte von Attributen einzelner Menüpunkte überladen werden können.
3. Das Framework muss effektive Menü-Definitionen von Rollen cachen

5.2 Wiederverwendung von Abfrage-Definitionen

5.2.1 Zweck

Ein und die selbe Abfrage-Definition kann in mehreren Kontexten verwendet werden. Im einfachsten Fall gibt es mehrere Menü-Elemente, die ein und die selbe Abfrage-Definition verwenden, dabei jedoch unterschiedliche Darstellungsformen angeben.

Beispielsweise könnte die Abfrage-Definition zur Anzeige der Anzahl der Ausleihen pro Standort einmal als Balken- und einmal als Torten-Diagramm ausgegeben werden. Hierzu bräuchten sich die beiden Menü-Elemente nur in dem Attribut *Type* zu unterscheiden – einmal mit dem Wert *pie* und einmal mit *bar*. Die Abfrage-Definition wäre in beiden Fällen die selbe. Sämtliche Attribute der betreffenden Menü-Elemente können sich unterscheiden, während die verwendete Abfrage-Definition gleich bleiben könnte.

Eine weitergehende Wiederverwendung von Abfrage-Definitionen kommt durch eine Parametrierung ihrer Inhalte zustande.

Ein Beispiel dafür sind die beiden KPIs für offene und überfällige Ausleihen auf dem Dashboard *Bibliothek* (siehe *Screenshot 1: Menü und ein Dashboard der Beispielanwendung*). Beide zählen Ausleihen, unterscheiden sich jedoch in der Bedingung der gezählten Ausleihen und bzgl. des verwendeten Icons.

5.2.2 Aufbau

```
30      , { "Id":"RENTAL_COUNT_OPEN"
31        , "Type":"kpi"
32        , "Label":"Offene Ausleihen"
33        , "File":"RENTAL_COUNT.query"
34        , "Column":"2"
35        , "Description": [ "Ausleihen ohne Rückgabedatum" ]
36        , "Parameters":
37          [ {"key":"FilterSelect",    "value":"RENTAL.RETURN_DAY IS NULL"}
38          , {"key":"Icon",            "value":"icon-folder-open"}
39          ]
40        }
41      , { "Id":"RENTAL_COUNT_LATE"
42        , "Type":"kpi"
43        , "Label":"Überfällige Ausleihen"
44        , "File":"RENTAL_COUNT.query"
45        , "Column":"3"
46        , "Description":
47          [ "Ausleihen, die vor mehr als drei Wochen getätigt wurden,"
48          , "zu denen aber noch keine Rückgabe erfasst wurde"
49          ]
50        , "Parameters":
51          [ {"key":"FilterSelect",    "value":"RENTAL.RETURN_DAY IS NULL
52                                          AND RENTAL.RENTAL_DAY < &dbdate - 21"
53            }
54          , {"key":"Icon",            "value":"icon-calendar5"}
55          ]
56        }
```

Definitionsbeispiel 12: Wiederverwendung einer Abfrage-Definition

Da beide der o. g. Menü-Elemente im Grunde das selbe bewirken (Ausleihen zählen), wird für beide dieselbe Abfrage-Definitionsdatei (*RENTAL_COUNT.query*) verwendet. Sie haben jedoch unterschiedliche Filter-Bedingungen im Eintrag mit dem Schlüssel *FilterSelect* des Attributs *Parameters*, bei dem es sich um ein Array mit Key/Value-Paaren handelt.

5.2.3 Auswertung

Besitzt ein Menü-Element ein Parameter-Paar mit dem Schlüssel *FilterSelect*, wird das an Hand der Abfrage-Definition generierte Statement um die dortige Filterbedingung ergänzt. Auf diese Weise lassen sich Abfrage-Definition mehrfach, jedoch mit unterschiedlichen Zusatzbedingungen nutzen – hier einmal mit der Bedingung, dass der Rückgabetag nicht erfasst ist und einmal zusätzlich mit der Bedingung, dass die Ausleihe schon mehr als drei Wochen her sein muss.

Während der hierfür verwendete Schlüssel *FilterSelect* ein im Framework reserviertes Wort für eben diesen Zweck ist, können weitere Schlüssel verwendet

werden, um beliebige Inhalte der Abfrage-Definition durch einen über das jeweilige Menü-Element gesteuerten Wert zu ersetzen.

In obigem Beispiel wird für beide Menü-Elemente ein Key/Value-Paar mit dem Schlüssel *Icon* festgelegt. Hierbei handelt es sich nicht um das *Icon*-Attribut des Menü-Elements selbst, sondern um einen als Parameter an die Abfrage-Definition weitergegebenen Icon-Namen. Die Verwendung von Icons in Abfrage-Definitionen wird im Kapitel *Kennzahlen* beschrieben.

Beim Lesen und Auswerten einer Abfrage-Definition wird geschaut, ob durch das jeweilige Menü-Element Parameter mitgegeben werden. Pro Parameter wird dessen Schlüssel (eingerahmt in Platzhaltertrennzeichen #< und #> – also #<*Schlüssel*>#) in der Abfrage-Definitionsdatei gesucht und durch den Parameterwert ersetzt.

5.2.4 Anforderung an das Framework

Um Abfragedefinition mehrfach wiederverwenden zu können, muss das Framework folgende Definitionsmöglichkeiten unterstützen:

1. Mehrere Menü-Elemente müssen die selbe Abfrage-Definitionsdatei mit geänderten Menü-Attributen verwenden können.
2. Zusätzliche Filter-Bedingungen müssen über die Menü-Definition mitgegeben werden können.
3. Es müssen beliebige Parameter angegeben werden können, mit deren Werten Platzhalter im Inhalt von Abfrage-Definitionsdateien ersetzt werden.

5.3 Wiederverwendung von Filter-Definitionen

5.3.1 Zweck

Auch Filter-Definitionen müssen sowohl wiederverwendet als auch abweichend zur Namenskonvention verwendet werden können.

Beispielsweise macht es Sinn, den Menüpunkt *Stammdaten/Benutzer* für die Rolle *MEMBER* umzuwidmen. Anwender dieser Rolle könnten in der Benutzer-Liste ohnehin nur ihre eigenen Daten sehen, da die Filter-Definition auf die Tabelle *MEMBER* für die Rolle *MEMBER* auf den eigenen Datensatz einschränkt. Den Eigenen Datensatz können Anwender dieser Rolle aber bereits in dem Dashboard sehen, der oben in „Meine Daten“ umbenannt wurde und sich bei Programmstart als erstes öffnet. Ein weiterer Menüpunkt, in dem man seinen eigenen Stammdatensatz sehen kann, wäre also überflüssig.

Stattdessen wäre an dieser Stelle jedoch eine Liste der Ansprechpartner des eigenen Standorts sinnvoll – also der Benutzer der Rolle *EMPLOYEE*, die dem selben Standort zugeordnet sind.

Problem dabei ist jedoch, dass für die Rolle *MEMBER* die Tabelle *MEMBER* an Hand einer Filter-Definition auf den eigenen Datensatz beschränkt ist. Gemäß dieser Filter-Definition sähe also ein Anwender der Rolle *MEMBER* gar keine Datensätze von Benutzern mit der Rolle *EMPLOYEE*.

5.3.2 Aufbau

Hierfür wird der Aufbau der Dateinamenskonvention ergänzt. Statt wie bisher aus zwei Namensbestandteilen (Rolle und Tabelle) kann ein Dateiname für eine Filter-Definition aus drei Namensbestandteilen bestehen (Rolle, Tabelle und Alias), so dass eine Tabelle anders gefiltert wird, wenn sie mit einem bestimmten Alias in der Abfrage-Definition angegeben ist.

Während die in Abschnitt *3.4.4* beschriebene Filter-Datei mit dem Dateinamen *MEMBER.MEMBER.query* für die Rolle *MEMBER* (erster Namensbestandteil) die Tabelle *MEMBER* (zweiter Namensbestandteil) so filtert, dass Anwender dieser Rolle nur ihren eigenen Datensatz aus dieser Tabelle sehen dürfen, kann die Filter-Datei mit dem Dateinamen *MEMBER.MEMBER.EMPLOYEE.query* eine davon abweichende Filter-Bedingung angeben, die dann greifen soll, wenn Anwender dieser Rolle diese Tabelle mit einer Abfrage-Definition selektieren, bei der diese Tabelle den Alias *EMPLOYEE* erhält.

Hierdurch erhalten Filter-Definition die Möglichkeit, vom inhaltlichen Kontext abzuhängen – ein Mitglied darf zwar die Daten anderer Mitglieder nicht sehen, aber er darf die Daten von Bibliotheks-Mitarbeitern sehen, die in der selben Tabelle gehalten werden, dies aber nur dann, wenn die Daten der betreffenden Tabelle über den Alias eine entsprechende Semantik erhalten.

Die speziellere Filter-Definition *MEMBER.MEMBER.EMPLOYEE.query* überlädt die allgemeine Filter-Definition *MEMBER.MEMBER.query*.

In obigem Fall existiert bereits eine Filter-Definition, die Daten aus der Tabelle *MEMBER* auf diejenigen Zeilen beschränkt, die zum selben Standort gehören – nämlich in Form der Filter-Definitionsdatei *EMPLOYEE.MEMBER.query*, die verwendet wird, um Anwendern der Rolle *EMPLOYEE* nur diejenigen Daten der Tabelle *MEMBER* zu zeigen, auf die sie zugreifen dürfen.

Diese bereits bestehende Filter-Definition der Rolle *EMPLOYEE* kann über die Zuordnungsdatei der Rolle *MEMBER* wiederverwendet werden, indem definiert wird, dass an Stelle der nicht vorhandenen Filter-Definitionsdatei *MEMBER.MEMBER.EMPLOYEE.query* die Datei *EMPLOYEE.MEMBER.query* verwendet werden soll.

```
1 [ { "key":"MEMBER.menu",                    "value":"ADMIN.menu"}
2 , { "key":"MEMBER.MEMBER.EMPLOYEE.query", "value":"EMPLOYEE.MEMBER.query"}
3 ]
```

Definitionsbeispiel 13: abweichende Filter-Definition in MEMBER.map

5.3.3 Auswertung

Beim Generieren von SQL-Statements und dem damit verbundenen Generieren von Filter-Bedingungen wird also zunächst geschaut, ob es für die Rolle des abfragenden Anwenders und die jeweilige Tabelle eine spezielle Filter-Definition für den Alias gibt, mit der die Tabelle ins Statement aufgenommen wird. Dabei ist es unerheblich, ob eine entsprechende Filter-Datei tatsächlich existiert oder ob über eine Zuordnungsdatei definiert ist, dass an ihrer Stelle eine andere Datei verwendet werden soll.

Existiert keine spezielle Filter-Definition für den jeweiligen Alias, wird eine nicht alias-bezogene Filter-Definition gesucht, wobei ebenfalls zunächst geschaut wird, ob über die Zuordnungsdatei eine abweichende Datei angegeben wird oder eine Datei laut Namenskonvention existiert.

5.3.4 Anforderungen an das Framework

Um Filterdefinitionen überladen und/oder wiederverwenden zu können,

1. muss definiert werden können, dass für bestimmte Aliase speziellere Filter gelten, und
2. in Zuordnungsdateien müssen auch Dateiersetzungen für Filter-Definitionsdateien angegeben werden können .

6 Extension-Points

Mit der bisher beschriebenen Funktionalität des Frameworks lassen sich Anwendungen wie die oben vorgestellte Beispielanwendung, die den Funktionsumfang der in [2] vorgestellten Bibliotheksverwaltung zuzüglich eines Berechtigungskonzepts und unterschiedlicher Auswertungen umfasst, als reine No-Code-Variante realisieren.

Das Ergebnis ist jedoch lediglich ein elektronischer Karteikasten, in dem Daten zu Büchern, Bibliotheksbenutzern und deren Ausleihen abgelegt und entsprechend einem Berechtigungskonzept abgerufen werden können.

Die bislang vorgestellten Konzepte nehmen der Entwicklerin in No-Code-Manier Schreibarbeit für CRUD-Operationen, Auswertungen und Berechtigung ab.

Um jedoch echte Geschäftslogik in die Anwendung einzubringen, reichen die vorgestellten Definitionen nicht aus. Daher sieht das Framework sowohl frontend- als auch backendseitig Erweiterungspunkte, sog. Extension-Points, vor, die den Aufruf von individuell programmierter Funktionalität in Form von Callback-Funktionen ermöglichen.

Diese Erweiterungspunkte stellen den Übergang vom No-Code- zum Low-Code-Framework dar.

6.1 Frontendseitige Extension-Points

6.1.1 Zweck

Das Framework besitzt ein in JavaScript entwickeltes Frontend. Dementsprechend sind die frontendseitigen Extension-Points ebenfalls in Form von JavaScript-Funktionen zu implementieren.

Besonderes Gewicht fällt hierbei den Editoren zu. Während der Erfassung eines Datensatzes wird u. a. die Validierung der Eingaben durch Geschäftslogik bestimmt.

Einfache Validierungen wie beispielsweise Pflichtfeldregeln, Wertebereiche, Datentypen, Eingabelängen oder einfache Beziehungen zwischen Eingabefeldern, wie z. B., dass eine Buchrückgabe nicht vor ihrer Ausleihe erfolgen kann, können freilich definitorisch gelöst werden (als Beispiel für ein Datumsfeld, dessen Minimum-Wert durch ein anderes Datumsfeld bestimmt ist, siehe Zeile 17 des Ausleihe-Editors in *Definitionsbeispiel 7*).

Anwendungsspezifische Validierungen, Berechnungen, Vorbelegungen und komplizierte Interdependenzen zwischen den Bedienelementen erfordern jedoch häufig individuellen Code.

6.1.2 Aufbau

Zur Integration von individuell programmiertem Code kann pro Editor eine JavaScript-Datei angegeben werden, die eine Event-Manager-Klasse mit Event-Handlern für diejenigen Ereignisse enthält, auf die mit Geschäftslogik reagiert werden soll.

Bei den berücksichtigten Ereignissen orientiert sich das Framework am Lebenszyklus eines Editors (siehe auch Abschnitt *Lebenszyklus eines Editors*) und an Ereignissen von datenselektierenden GUI-Komponenten.

Für sämtliche Ereignisse gibt es jeweils sowohl einen Extension-Point, der vor dem Ereignis ausgeführt wird, als auch einen, der danach ausgeführt wird.

Entsprechende Callback-Funktionen für die in nachstehender Tabelle genannten Ereignisse können, müssen jedoch nicht existieren.

Ereignis	**Verwendung**
Laden des Editors	Hier können Vorbelegungen vorgenommen werden oder auch zusätzliche frontendseitige Event-Handler, die nicht vom Framework berücksichtigte Ereignisse betreffen.
Laden des bearbeiteten Datensatzes	Sobald der zu editierende Datensatz geladen ist, mögen Berechnungen und weitere Vorbelegungen erforderlich sein.
Laden der Daten datenselektierender Komponenten	Das Laden der Daten datenselektierender GUI-Komponenten erfolgt asynchron bei Bedarf. Master-Detail-Tabellen aus einem beim Start des Editors nicht geöffneten Karteireiter z. B. brauchen erst geladen zu werden, wenn dieser angewählt wird. Sobald die Daten geladen wurden, mag Anwendungslogik mit diesen Daten ausgeführt werden müssen.
Auswahl eines Datensatzes in einer Combobox oder einem SDT	Wenn in einer Combobox oder einem SDT ein Datensatz ausgewählt wird, mögen Berechnungen durchgeführt oder abhängige Daten geladen werden müssen.
Vorbereitung der Speicherung	Vor der Speicherung eines Datensatzes greifen etwaige definitorisch festgelegte Validierungen. Unmittelbar davor oder danach mag es erforderlich sein, weitere Validierungen vorzunehmen.

Tabelle 6: Ereignisse für frontendseitige Callbacks

Nachstehendes Beispiel zeigt eine Event-Manager-Klasse mit Namen BUX.BUCH_EDIT für den Buch-Editor, die in diesem Fall genau eine Callback-Funktion enthält – und zwar die, die nach dem Aufbau des Bearbeitungsdialogs und seiner HTML-Elemente aufgerufen wird, was ein geeigneter Zeitpunkt ist, um beispielsweise Validatoren für einzelne Bedienelemente festzulegen.

Hier wird unter Zuhilfenahme der Bibliothek jQuery Validate [30] ein einfacher Validator namens `validate_book_isbn` für ISBN definiert und dem Eingabefeld `dfsISBN` zugewiesen, so dass bei jeder Änderung in diesem Feld mittels der Funktion `isValidIsbn` geprüft wird, ob es sich bei der Eingabe um eine gültige ISBN handelt. Falls dies nicht der Fall ist, wird eine Fehlermeldung unterhalb des Eingabefeldes platziert und ein Speichern des Datensatzes blockiert.

```
1  var BUX = BUX || {};              // Namespace für BUX
2
3  BUX.BOOK_EDIT = function() {      // Event-Manager-Klasse für den Buch-Editor
4    /* Callback-Methode, die nach dem Seitenaufruf aufgerufen wird
5     */
6    this.loadFinished = function() {
7      // Validator für ISBN erzeugen
8      $.validator.addMethod('validate_book_isbn', isValidIsbn, 'ungültige ISBN');
9
10     // Validator an das ISBN-Feld heften
11     $('#dfsISBN').rules('add', 'validate_book_isbn');
12     return true;
13   }
14 }
15
16 window.udqEditorManager.EventManager = new BUX.BOOK_EDIT();
17 //# sourceURL=BOOK_EDIT.js
```

Quellcode 1: Event-Manager-Klasse in BOOK_EDIT.js mit Callback-Funktion

Der hier verwendete Validierungsmechanismus ist nicht Gegenstand der Ausführung dieses Dokuments, weshalb auch nicht näher auf den Algorithmus zur Prüfung von ISBN eingegangen wird. An dieser Stelle soll lediglich darauf hingewiesen werden, dass anwendungsspezifische Funktionalität dieser und natürlich auch komplexerer Art über Callback-Funktionen eingebaut werden kann.

6.1.3 Auswertung

Der Editor-Manager des Frameworks, der die Ausführungskontrolle von Editoren regelt, bekommt in seinem Attribut EventManager ein neu instantiiertes Objekt der Event-Manager-Klasse zugewiesen (Zeile 16 des obigen Beispiels). Enthält die Event-Manager-Klasse eine Callback-Funktion für das jeweilige Ereignis, so führt der Editor-Manager diese aus und gibt ggf. entsprechende Parameter mit. Als Parameter für Ereignisse, die sich auf datenselektierende GUI-Komponenten beziehen, wird beispielsweise die jeweilige Komponente übergeben.

Bei der Vorbereitung der Speicherung kann die Callback-Funktion einen Rückgabewert haben, der besagt, dass die Speicherung (z. B. auf Grund einer fehlgeschlagenen Validierung) abgebrochen werden soll.

Da es sich bei diesem Dokument nicht um ein Tutorial für ein bestimmtes Framework handelt, sondern nur um eine Anforderungsbeschreibung an eine bestimmte Art von Framework, wird nicht näher darauf eingegangen, wie die Callback-Funktionen heißen sollen oder welche Parameter sie genau erhalten. Auch kann obige Liste der Ereignisse bei der Entwicklung eines entsprechenden Frameworks ergänzt werden.

Die Einsatzmöglichkeiten solcher Callback-Funktionen erschöpfen sich nicht in mehr oder weniger komplexen Validierungen, sondern können auch genutzt werden, um eine komplexe GUI aufzubauen. Auch, wenn hier von Editoren die Rede ist, kann es sich bei der aufgerufenen Seite auch um eine Benutzerführung handeln, die nicht der Bearbeitung eines Datensatzes dient und daher kein Editor im engeren Sinne ist. Die hier vorgestellten Prinzipien lassen sich jedoch analog anwenden.

6.1.4 Anforderungen an das Framework

Das Framework muss mindestens bei den o. g. Ereignissen

1. entsprechende Callback-Funktionen aufrufen.
2. Es muss definiert werden können, wo sich die Callback-Funktionen befinden
3. Die Callback-Funktionen müssen optional sein.

6.2 Backendseitige Extension-Points

6.2.1 Zweck

Frontendseitige Validierung sorgt für eine sachgemäße Benutzerführung, kann jedoch die Korrektheit der an das Backend gesendeten Daten nicht sicherstellen, da die REST-Services des Backends nicht nur von der vorgesehenen GUI angesprochen werden können, sondern auch direkt und somit an der GUI und der darin implementierten Logik vorbei.

Bevor also vom Client stammende Daten in die Datenbank aufgenommen werden, müssen sie im Falle einer Web-Anwendung backendseitig validiert werden.

Dies gilt für die gesamte Geschäftslogik. Sie kann zur Unterstützung des Anwenders – etwa zur Vermeidung von Fehleingaben – zusätzlich im Frontend enthalten sein, das Sicherstellen der Einhaltung von Geschäftslogik muss jedoch am Backend geschehen.

Manche Regeln aus der Geschäftslogik lassen sich über das Rollenkonzept bzw. über Filter-Definitionen lösen – beispielsweise, dass Anwender nur Bücher entsprechend ihrer Altersbeschränkung sehen dürfen. Entsprechende Filter können definiert werden, sofern die dafür benötigten Merkmale an den Datensätzen für Anwender und Bücher hinterlegt werden können, was das Datenmodell der Beispielanwendung jedoch nicht vorsieht.

Manche Regeln erfordern jedoch individuelle Programmierung.

6.2.2 Aufbau

Hierzu können in den Abfrage-Definitionen eines Editors backendseitige Callback-Funktionen angegeben werden, die während des Speichervorgangs ausgeführt werden – sowohl solche, die vor dem aus der Abfrage-Definition generierten Speicher-Statement ausgeführt werden, als auch solche, die danach ausgeführt werden.

6.2.3 Auswertung

Das Framework prüft bei relevanten Backend-Ereignissen (Selektieren, Speichern, Löschen), ob in der Abfrage-Definition Callback-Funktionen angegeben sind, die vor bzw. nach dem vom Framework generierten Statement ausgeführt werden sollen. Sofern diese angegeben sind, wird geprüft, ob die Ausführung dieser Funktionen entsprechend der Menü-Definition der Rolle des Anwenders erlaubt ist (Menü-Definitionen können hierzu Einträge vom Typ *script* enthalten,

deren ID der Name einer Callback-Funktion ist und deren *File*-Attribut das Modul angibt, in dem die Funktion definiert ist). Ist die Ausführung erlaubt, wird die jeweilige Callback-Funktion aufgerufen. Andernfalls wird die Transaktion abgebrochen.

Solche Callback-Funktionen können auch, aber nicht nur für die Validierung von Eingaben genutzt werden. Andere Einsatzszenarien ergeben sich, wenn die Geschäftslogik erfordert, dass mehr ausgeführt werden soll als lediglich das aus der Abfrage-Definition generierte Speicher-Statement. Dies kann Aktualisierungen in weiteren Tabellen oder berechnete Werte betreffen.

An Stelle der einfachen Eingabe-Dialoge der hier vorgestellten Beispielanwendung sind komplexe Assistenten denkbar, die den Benutzer in mehreren Dialogschritten durch einen Prozess führen und Eingaben aufnehmen, die letztlich in mehreren Datenbanktabellen gespeichert werden müssen. Hier reicht ein aus der Abfrage-Definition generiertes Statement nicht aus, um die in mehreren Dialogschritten des Frontends aufgenommenen und an das Backend gesendeten Daten in mehreren Tabellen abzulegen.

In solch einem Fall kann mit den hier vorgestellten Callback-Funktionen, die vor oder nach der Ausführung des generierten Statements, jedoch innerhalb der selben Transaktion, aufgerufen werden, die geforderte komplexe Businesslogik realisiert werden.

Zur Veranschaulichung folgt hier jedoch eine simple Callback-Funktion, die lediglich der backendseitigen Überprüfung der eingegebenen ISBN dient. Sie wird beim Speichern mittels des Buch-Editors vor der Ausführung des an Hand der Abfrage-Definition des Editors generierten UPDATE- bzw. INSERT-Statement aufgerufen.

Die Callback-Funktion kann zunächst relevante Informationen aus dem Request abholen – bzw. aus dem Framework-Kontext, der den Request abarbeitet. Dadurch kann die Eingabe des aufrufenden Editors abgefragt und in einer Variablen (hier `Input`) abgelegt werden. Die Eingabe enthält nicht nur den Inhalt des validierten Feldes, sondern die Werte aller Felder des Editors, die über Bind-Klassen an die Abfrage-Definition des Editors gebunden sind, so dass auch Interdependenzen zwischen den Eingabefeldern geprüft werden können.

Mit diesen Informationen führt die Callback-Funktion die Prüfung durch und gibt ein Dictionary zurück, in dem ein Status und im Falle eines Fehlers auch eine entsprechende Fehlermeldung enthalten ist.

In obigem Beispiel erfolgt die eigentliche Prüfung der ISBN mit einer in Zeile 20 aufgerufenen Funktion, die jedoch nicht Gegenstand dieser Betrachtung sein soll.

```python
import re
from ..udq.udqUtils import getSave
from ..udq.udqEnvironment import UdqEnvironment

def validate_book_isbn(**kwargs):
    """ Validiert eine ISBN
    """

    # Relevante Infos aus dem Request holen
    Retval        = UdqEnvironment.StandardResponse()
    context       = kwargs.get('context')
    BodyParams    = kwargs.get('bodyparams')
    Input         = getSave(BodyParams, 'Input')
    isbn          = context.ItemGetValue(Input, {'Alias':'ISBN'})

    # ISBN auswerten
    if not isbn:
        Retval['status'] = 'error'
        Retval['readable'] = 'Die ISBN darf nicht leer sein'
    elif not is_valid_isbn(isbn):
        Retval['status'] = 'error'
        Retval['readable'] = 'Die ISBN %0% ist ungültig.'
        Retval['params'] = [isbn]
    else:
        Retval['status'] = 'ok'
    return Retval
```

Quellcode 2: Backenseitige Callback-Funktion in BOOK_EDIT.py

6.2.4 Synergieeffekte zwischen Frontend- und Backend-Validierung

Da die Validierung, wie oben beschrieben, ohnehin auch backendseitig erfolgen muss, bietet es sich an, backendseitig implementierte Validierungen auch frontendseitig zu nutzen, damit nicht ein und die selbe Prüfroutine zwei mal programmiert werden muss – einmal für das Frontend in JavaScript; und einmal für das Backend in dessen Programmiersprache (hier Python). Eine solche Doppelprogrammierung ist nicht nur aufwändig, sondern birgt auch Fehlerpotenzial, wenn beide in unterschiedlichen Programmiersprachen entwickelte Prüfungen nicht exakt identisch arbeiten.

Zur frontendseitigen Wiederverwendung der Backend-Validierung ist lediglich zu realisieren, dass eine Eingabeprüfung sofort nach Verlassen des jeweiligen Eingabefeldes erfolgt und nicht erst beim Speichern des Datensatzes.

Das Framework sieht hierfür vor, für das Eingabefeld in dessen HTML-Definition ein Attribut namens *udq-validator* zu setzen, das die backendseitige Validierungsfunktion angibt (Zeile 28 in nachstehendem HTML-Ausschnitt). Bei Feldern, wo dieses Attribut gesetzt ist, wird beim Verlassen des Feldes automatisch ein Request gesendet, der die jeweils angegebene backendseitige Prüfung ausführt.

```
26    <div class="form-group breakpoint-input">
27      <input type="text" id="dfsISBN" class="form-control dfString Bind-String"
28       maxlength="24" required udq-validator="validate_book_isbn">
29      <label class="form-control-placeholder mlLabel" for="dfsISBN">ISBN</label>
30    </div>
```

Definitionsbeispiel 14: Angabe der Backend-Validator-Funktion im Frontend

Aus Anwenderperspektive verhält sich diese Form der Validierung wie die vollständig frontendseitig implementierte Validierung aus *Quellcode 1*: Beim Verlassen des Feldes wird validiert und m Falle eines Verstoßes gegen die Eingaberegeln erscheint eine Fehlermeldung.

Die Verwendung von Backend-Validatoren auch für frontendseitig initiierte Validierungen hat gegenüber der vollständig frontendseitigen Validierung den Vorteil, dass die Validierung nur einmal programmiert werden muss. Jedoch hat sie auch den Nachteil, dass für jede einzelne Prüfung ein Request vom Client zum Server geht. Einfache Frontend-Validierungen, wie die Prüfung, ob sich die Eingabe um eine gültige ISBN-Nummer handelt, die ausschließlich auf der Eingabe operieren und keine Daten vom Server benötigen, sind daher vorzugsweise vollständig frontendseitig vorzunehmen, sofern die Doppelprogrammierung sich in vertretbarem Rahmen bewegt.

6.2.5 Weitere Anwendungsgebiete backendseitiger Callback-Funktionen

Wenn allerdings Daten des Servers benötigt werden, kommt man nicht umhin, einen Request an den Server zu senden. Beispiele hierfür sind Prüfungen, ob eine einzugebende Nummer bereits für einen anderen Datensatz vergeben ist.

In der Beispiel-Bibliotheksanwendung aus [2] ist die Benutzer-Nummer ein Eingabefeld. Es wird dort auch sichergestellt, dass sie systemweit eindeutig ist. Auch dies kann mit dem oben vorgestellten Verfahren realisiert werden. Da es für einen Anwender jedoch kaum möglich ist, eine solche Nummer eindeutig zu vergeben, sollte sie besser systemseitig vergeben werden. Eine solche systemseitige Vergabe einer eindeutigen Nummer – z. B. inkl. einer Bildungsvorschrift – kann mit einer backendseitigen Callback-Funktion realisiert werden.

Ein praxistauglicheres Beispiel für eine Eindeutigkeitsprüfung ist die Eingabe eines amtlichen Fahrzeugkennzeichens in einem Fuhrparkmanagementsystem. Das Kennzeichen kann nicht vom Fuhrparkmanagementsystem selbst generiert werden, sondern stammt aus einer externen Quelle und wird bei der manuellen Fahrzeugerfassung vom Anwender eingegeben. Dabei ist sicherzustellen, dass das Kennzeichen relativ eindeutig ist. Die Eindeutigkeit wird hier insofern relativiert, als es natürlich vorkommen kann, dass mehrere Fahrzeuge für einander nicht überschneidende Zeiträume das selbe Kenneichen haben.

Für solche Zwecke eignen sich die oben vorgestellten backendseitigen Callback-Funktionen. Die Geschäftslogik, die dabei zu berücksichtigen ist, muss individuell programmiert werden und kann eben auf Grund der in der Geschäftslogik begründeten Relativierung der Eindeutigkeit nur bedingt durch Datenbank-Constraints realisiert werden. Schließlich reicht es zur Überprüfung der Überschneidungsfreiheit der Gültigkeitszeiträume von Fahrzeugkennzeichen nicht aus, lediglich die Eindeutigkeit der Kombination aus Kennzeichen, Erstzulassung und Abmeldung sicherzustellen. Außerdem mag die Geschäftslogik in diese Prüfung auch Attribute von anderen Tabellen als der Fahrzeugtabelle einbeziehen, so dass die deklarativen Möglichkeiten von DDL überschritten werden mögen.

Das oben angesprochene Beispiel der Benutzer-Nummer innerhalb der Bibliotheksanwendung lässt sich ebenfalls mittels backendseitigen Callbacks realisieren – nur eben praktischerweise nicht wie in [2] mittels einer Validierung, sondern in Form einer automatischen Nummernvergabe.

Hierzu ist eine backendseitige Callback-Funktion zu implementieren, die bei der Neuanlage eines Benutzers mit dem Benutzer-Editor vor der Ausführung des durch das Framework auf Basis der Abfrage-Definition generierten INSERT-Statements eine Nummer für den Benutzer erzeugt und der Eingabe für das generierte INSERT-Statement hinzufügt.

6.2.6 Anforderung an das Framework

Das Framework muss die Möglichkeit bieten,

1. vor und nach der Ausführung der auf Basis von Abfrage-Definition generierten Statements individuell programmierte Funktionen aufzurufen.
2. Diese Funktionen müssen innerhalb der selben Transaktion ausgeführt werden können wie die generierten Statements.
3. Sie müssen geeignet sein, bei Bedarf die Transaktion abzubrechen.
4. Sie müssen Zugriff auf die komplette Eingabe erhalten.
5. Vor dem generierten Statement ausgeführte Callback-Funktionen müssen die Eingabe anpassen oder erweitern können, so dass diese Angaben in die Generierung einfließen.
6. Nach dem generierten Statement ausgeführte Callback-Funktionen müssen Kenntnis der IDs erlangen, die für die oder bei der Ausführung des generierten Statements entstehen (etwa, wenn das generierte Statement ein INSERT für eine Tabelle mit einem datenbankseitig generierten Primärschlüssel ist).

7 Evaluation

Ein nach den hier vorgestellten Anforderungen erstelltes Framework ermöglicht es, eine Web-basierte Datenbankanwendung inklusive Rollenkonzept zu entwickeln.

7.1 Effizienzsteigerung bei CRUD-Operationen

Typische CRUD-Funktionalität kann dabei an Hand von Definitionen und ohne Programmierung realisiert werden. Dies kann wesentlich effizienter erfolgen als dies beispielsweise mit dem erst im Januar 2020 final herausgegebenen Java MVC Framework [3] möglich ist.

Um diese These zu stützen wurde in diesem Papier eine in einem Lehrbuch für Java MVC Framework [2] beschriebene und dort ca. 1.500 Zeilen Quellcode umfassende Beispiel-Anwendung an Hand von Definitionen für das hier vorgestellte Framework nachimplementiert.

Die Definitionen, die den Funktionsumfang der Beispielanwendung aus [2] realisieren, umfassen

- ca. 20 Zeilen für eine Menü-Definition,
- drei Abfrage-Definitionen mit insgesamt ca. 210 Zeilen,
- drei HTML-Dateien mit insgesamt ca. 100 Zeilen[14]

Der im Falle dieser Beispielanwendung gemessen an Schreibarbeit in Zeilen ca. um Faktor fünf höhere Aufwand der Lösung mit Java MVC lässt sich mit einer großen Menge an technisch bedingtem „boilerplate“ Code begründen, der wenig Kreativität erfordert.

Dagegen ermöglicht der auf Definitionen basierende Ansatz des hier vorgestellten Frameworks der Entwicklerin, sich auf die inhaltlichen Aspekte ihrer Software zu konzentrieren und festzulegen, wer welche Daten wie und in welchem Umfang bearbeiten darf.

[14] Insgesamt besitzt die unter https://bux.fleetweb-service.de bereitgestellte Beispiel-Anwendung einen größeren Funktionsumfang als die in [2] beschriebene und benötigt dafür weitere Definitionen. Die o. g. Zeilenanzahlen beziehen sich ausschließlich auf den Anteil, der die Funktionalität aus [2] nachbildet.

7.2 Qualitäts- und Aufwandsvorteile gegenüber Reports

Die in den Dashboards gezeigten Auswertungsmöglichkeiten könnten auch mittels Reports realisiert werden, bieten diesen gegenüber jedoch insofern einen qualitativen Vorteil, als sie interaktiv sind – sprich: der Anwender kann durch die Drilldown-Möglichkeiten tiefer in die Materie einsteigen. Er kann, sofern er über nötige Berechtigungen verfügt, auch Änderungen an den zugrundeliegenden Daten vornehmen, was in Berichten üblicherweise nicht möglich ist.

Auch bzgl. des Aufwands lässt sich ein Vorteil gegenüber Reports feststellen, da letztere i. d. R. einen hohen Anspruch an das Layout haben, der jeweils Aufwand bedeutet. Allein das Anordnen von Informationen, so dass ein praktikabler Seitenwechsel entsteht, kann zeitraubend sein. Diesen Aufwand braucht man weder für Tabellen, KPIs noch für Businessgrafiken zu betreiben, da diese von Hause aus (d. h. durch das Framework) stets dem erwarteten Layout entsprechen.

Dennoch verbleiben Einsatzszenarien für Report-Generatoren – etwa für die Erstellung von Rechnungen und anderen Formularen.

7.3 Von No-Code nach Low-Code

Des Weiteren wurde hier gezeigt, dass ein solches Framework Möglichkeiten zur Integration von individueller Programmierung bieten kann, so dass man einerseits die Vorteile von No-Code-Komponenten nutzen kann, und andererseits die Flexibilität der Programmierung behält, weshalb hier von einem Low-Code-Framework gesprochen werden kann.

7.4 Nicht pauschal quantifizierbare Gesamtersparnis

Der Anteil an erforderlicher Programmierung variiert je nach geforderter Businesslogik, weshalb der oben genannte Aufwandsunterschied um Faktor fünf nicht als repräsentativ betrachtet werden kann. Wenn die in diesem Framework sehr gut unterstützten CRUD-Operationen nur einen geringen Anteil er Software ausmachen, sinkt entsprechend die Aufwandsersparnis.

Zudem muss berücksichtigt werden, dass obiger Aufwandsvergleich an der Anzahl der zu schreibenden Zeilen gemessen wurde und dass die Zeilenanzahl anerkanntermaßen kein geeignetes Maß für die Leistung einer Entwicklerin darstellt und somit auch nicht den Gesamtaufwand ausdrückt.

Wenige Zeilen gut durchdachten Codes oder entsprechender Definitionen können mehr Aufwand erfordern als viele Zeilen „boilerplate“ Code, der ggf. sogar

generiert wurde und dessen Erzeugung möglicherweise weniger Aufwand erfordert als dessen Wartung.

Außerdem umfasst die Softwareentwicklung neben der Erstellung von Code und/oder Definitionen viele weitere Aspekte, die in dieser Arbeit nicht betrachtet wurden. Hier sind u. a. die inhaltliche Durchdringung der Anforderungen, die Datenmodellierung, die Planung und Durchführung von Tests und die Fehlerbehebung zu nennen.

Daher sind Aussagen zu Effizienzsteigerungen bezogen auf den Gesamtaufwand mit Vorsicht zu genießen.

Eine Quantifizierung der Effizienzsteigerung durch das hier vorgestellte Framework wird hier also ausdrücklich nicht vorgenommen und muss stattdessen mit Blick auf die zu erstellende Software betrachtet werden.

Fest steht jedoch, dass CRUD-Operationen mit einem solchen Framework sehr effizient realisiert werden können. Analoges gilt für Auswertungen und das Rollenkonzept.

7.5 Framework-unabhängige Herangehensweise

Die hier vorgestellten Prinzipien lassen sich auch in Software integrieren, die nicht überwiegend aus Definitionen besteht.

So ließe sich beispielsweise die im Kapitel *Referentielle Integrität beim Löschen* beschriebene Lösung auch unabhängig vom Rest des hier vorgestellten Frameworks in einer ansonsten nicht definitionsgetriebenen Anwendung verwenden.

Die definitionsgetriebene Softwareentwicklung ist, wie im zweiten Kapitel beschrieben, bei Reports gängige Praxis und kann in bestehenden Anwendungen verwendet werden – etwa dadurch, dass man zum Löschen eine zentrale definitionsbasierte Klasse verwendet. Diese sollte nicht lediglich Tabellenbeziehungen, wie sie in den Definitionen von OR-Mappern angegeben werden, sondern auch Geschäftslogik berücksichtigen, wie dies in o. g. Kapitel beschrieben wird.

Dies zeigt, dass der definitionsgetriebene Ansatz nicht von einem Framework abhängig ist, sondern auch punktuell angewandt werden kann. Letztlich ist der definitionsgetriebene Ansatz eine stark ausgeprägte Form der Parametrierung und daher auch außerhalb eines bestimmten Frameworks anwendbar.

Das gleiche gilt für das in diesem Papier angesprochene und unter dem Namen „Konvention vor Konfiguration“ bekannte Prinzip, das hier u. a. für Dateinamenskonventionen verwendet wird – etwa dass der Name einer Menü-Definitionsdatei

vom Rollen-Namen abgeleitet wird, oder dass die bloße Existenz einer Filter-Definitionsdatei für eine Kombination aus Rollen- und Tabellenname (ggf. mit Alias) dazu führt, dass ein Rollen-Filter bei Zugriffen auf eine Tabelle angewandt wird.

Die Konvention, dass HTML-Elemente durch Namensgleichheit an die Elemente einer Abfrage-Definition gebunden werden, ist ein Beispiel dafür, dass Definitionen es erleichtern, Konventionen zu automatisieren.

Die in Kapitel *Synergieeffekte zwischen Frontend- und Backend-Validierung* vorgestellte Konvention, für die Verwendung eines Validators lediglich im betroffenen HTML-Element ein Attribut zu setzen, zeigt, dass bestehende und weit verbreitete Techniken wie HTML für eigene deklarative Funktionalität offen sein können.

Neben der Verwendung eines nach den hier vorgestellten Prinzipien entwickelten Frameworks lohnt es sich also auch, in anders konzipierten Anwendungen punktuell einige der hier beschriebenen Herangehensweisen zu verwenden.

7.6 Bestehende Implementierung der hier vorgestellten Konzepte

Wie in der Einleitung erwähnt, gibt es bereits eine Implementierung eines Frameworks nach den hier beschriebenen Konzepten sowie mit Fleet+ Web eine Anwendung, die auf Basis dieses Frameworks erstellt wurde.

Bei Fleet+ Web handelt es sich um eine Fuhrparkmanagementsoftware, die sich bei gut 20 Unternehmen im produktiven Einsatz befindet – darunter Fuhrparkdienstleister, die mehrere Tausend Fahrzeuge damit verwalten.

Entstanden ist Fleet+ Web als Zusatzprodukt zum Fuhrparkmanagementsystem Fleet+, das von mehreren Hundert Kunden genutzt wird und für das der Autor dieser Abschlussarbeit seit dem Jahr 2000 die technische Verantwortung innehat.

Fleet+ ist einerseits über die Jahrzehnte funktional sehr umfangreich geworden und erfüllt damit sämtliche Belange der Branche, andererseits handelt es sich allerdings um eine zweischichtige Windows-Anwendung, was den standortübergreifenden Einsatz und damit auch die direkte Einbeziehung des Endkunden erschwert.

Ende des letzten Jahrzehnts bestand somit die Herausforderung, eine Cloud-Anwendung zu schaffen, die Fleet+ langfristig ablösen könnte. Eine solche Anwendung zu entwickeln ist jedoch angesichts des in über 20 Jahren gewachsenen Funktionsumfangs sehr aufwändig.

Um möglichst zügig mit einem MVP[15] an den Markt gehen zu können, das bereits einen Mehrwert bietet, konzipierte und entwickelte der Autor dieser Arbeit das hier vorgestellte Framework. Hierbei wurden zur schnelleren Umsetzung des Vorhabens definitionsgetriebene Vorgehensweisen, die es bei Fleet+ bereits gibt, in optimierter Form übernommen und fortentwickelt.

Beispielsweise gibt es in Fleet+ bereits Dashboards, deren Inhalte anhand von Abfrage-Definitionen generiert werden. Kernstück der Abfrage-Definitionen von Fleet+ sind SELECT-Statements, mit denen die gewünschten Daten selektiert werden.

Abweichend davon enthalten die Abfrage-Definitionen von Fleet+ Web aus den im Kapitel *Abfrage-Definitionen* im Abschnitt *Auswertung* genannten Gründen nicht das letztlich verwendete Statement, sondern dessen Bestandteile wie z. B. selektierte Spalten und betroffene Tabellen in einer leicht automatisiert zu interpretierenden Datenstruktur.

Eine weitere, die Entwicklung von Fleet+ Web beschleunigende Maßnahme war, dass das für Dashboards bewährte datengetriebene Verfahren auch bei den zahlreichen Browser-Fenstern (d. h. tabellarische Auswahllisten zum Aufruf eines zu bearbeitenden Datensatzes) angewandt wurde. Derartige Browser sind in Fleet+ noch fest verdrahtet programmiert, in Fleet+ Web dagegen vollständig definitionsgetrieben erstellt worden.

Nachdem zunächst ein einfacher Interpreter für Abfrage-Definitionen erstellt wurde, konnte der über Fleet+ Web angebotene Funktionsumfang durch Hinzufügen immer weiterer Abfrage-Definitionen leicht und schnell erweitert werden.

Gleichzeitig konnte durch die Web-Anwendung Fleet+ Web das realisiert werden, was in der zweischichtigen Windows-Anwendung Fleet+ nicht darstellbar war – nämlich die Einbeziehung weiterer Nutzergruppen, insbesondere der Kunden der Lizenznehmer von Fleet+.

Fuhrparkdienstleister wurden in die Lage versetzt, ihren Kunden, deren Fahrzeuge sie verwalten, eine Sicht auf einen Ausschnitt ihrer Daten zu gewähren. Dafür spielte ein flexibles Rollenkonzept eine zentrale Rolle, so dass jeder Endkunde gesichert nur Zugriff auf die Daten erhält, die er sehen und bearbeiten darf.

---

[15] MVP = minimum viable product – sprich: ein minimales existenzfähiges Produkt

Hierbei ist zu bedenken, dass Rollen und die von ihnen einsehbaren Daten nach sehr unterschiedlichen Kriterien voneinander abgegrenzt werden können müssen. Die Abgrenzung von Sichtbarkeitsbereichen ist so vielfältig wie die Organisationsstruktur der Kunden.

Durch das flexible Rollenkonzept von Fleet+ Web und die Einbeziehung neuer Nutzergruppen wurde von Anfang an ein Mehrwert geboten, ohne gleich den gesamten Funktionsumfang von Fleet+ abdecken zu müssen.

Als wesentliche Erfolgsfaktoren von Fleet+ Web können somit das mittels Menü- und Filter-Definitionen umgesetzte Rollenkonzept und die leichte Erweiterbarkeit durch zusätzliche Abfrage-Definitionen genannt werden.

8 Ausblick

In dieser Abschlussarbeit wird zwar eine konkrete Syntax für Definitionen verwendet, und es werden sowohl die Bedeutung als auch die Wirkungsweise konkreter Definitionen erörtert, diese orientieren sich jedoch an dem bereits vorhandenen Framework des Autors, das selbst nicht Gegenstand dieser Arbeit ist, sondern hier nur dem Zwecke dient, Anforderungen an ein solches Framework an Hand eines durchgängigen Anwendungsbeispiels veranschaulichen zu können.

Schwerpunkt dieser Arbeit sind jedoch diese Anforderungen selbst bzw. deren Beschreibung und die Vorstellung eines Konzepts für ein solches Framework.

In einem nächsten Schritt kann auf Basis dieser Anforderungen und dieses Konzepts eine Spezifikation erstellt werden, die sowohl Syntax als auch Semantik der hier beschriebenen Definitionstypen festschreibt. Eine solche Spezifikation sollte idealerweise unternehmensübergreifend abgestimmt werden, damit sie nicht nur den Belangen eines bestimmten Anwendungskontextes genügt, sondern einen weiter gefassten Einsatzbereich abdeckt.

Dies kann Gegenstand weiterer Forschungen über die Einsetzbarkeit eines solchen Frameworks sein, an deren Ende eine konkrete Spezifikation steht. Eine solche Spezifikation kann von einer Referenzimplementierung flankiert werden.

Neben diesem vertiefenden konzeptionellen Ausblick gibt es auch rein produktbezogene weitere Schritte. So sind beispielsweise die in dieser Abschlussarbeit vorgestellten Definitionsbeispiele allesamt in einem Texteditor erstellt worden. Dies ist in der täglichen Praxis des Autors und der seiner Teamkolleginnen und Teamkollegen, die gemeinsam eine umfangreiche, auf einem solchen Framework basierende Software erstellen, völlig ausreichend.

Für Citizen-Developers wäre jedoch eine visuelle Unterstützung zur Erstellung solcher Definitionen hilfreich. Denkbar wäre hier ein grafischer Editor für Abfrage-Definitionen nach dem Vorbild von Microsoft Access[16] oder Crystal Reports[17], mit dem man Tabellen in Beziehung setzen und auszugebende Spalten auswählen kann.

[16] Microsoft Access ist ein Anwendungsentwicklungswerkzeug von Microsoft [31], das einen grafischen Abfrage-Designer enthält [32].

[17] Crystal Reports ist ein Reportgenerator von SAP, der über einen grafischen Abfrage-Designer verfügt [33]

Aus dem Fundus an mittels eines solchen grafischen Editors erstellter Abfrage-Definitionen könnte sich der Citizen-Developer dann eine Menü-Definition zusammenstellen.

Im Ergebnis würden dabei Definitionen der in diesem Dokument vorgestellten Art entstehen, doch der Weg dorthin wäre komfortabler.

Dies wäre ein sinnvoller künftiger Schritt, um ein solches Framework für Citizen-Developers vermarkten zu können.

9 Fazit

Die digitale Transformation wird auf absehbare Zeit den Bedarf an Softwareunterstützung für immer mehr Geschäfts- und Alltagsprozesse erhöhen – und dies bei bereits jetzt bestehendem Fachkräftemangel. Gemeinsam mit dem anhaltenden Kostendruck erfordert dies eine stetige Optimierung des Entwicklungsprozesses.

Der hier vorgestellte definitionsgetriebene Absatz kann durch seine Effizienzsteigerung bei der Realisierung von CRUD-Operationen und Auswertungen einen Beitrag hierzu leisten.

Dabei spielt die Unterstützung eines einerseits ausgeklügelten und andererseits leicht beherrschbaren Rollenkonzepts eine wichtige Rolle, da die Ausweitung von Geschäftsprozessen zur immer stärkeren digitalen Einbindung des Kunden führt. Software wird nicht nur für den unternehmensinternen Gebrauch eingeführt, sondern auch zur Interaktion und zum Datenaustausch mit dem Kunden.

Statt Rechnungen in Papierform per Post oder etwas fortschrittlicher digital per E-Mail zu versenden, kann auf einen Versand gänzlich verzichtet werden, wenn man dem Kunden direkten Zugriff auf ein Portal gewährt, wo er jederzeit seine Rechnungen und die zugrundeliegenden Geschäftsvorfälle einsehen kann. Idealerweise kann er dort gewisse Arbeiten der Datenpflege (Namens- und Adressänderungen etc.) selbst vornehmen.

Dies erfordert ein Rollenkonzept, das die Daten der unterschiedlichen Kunden und anderer Akteure sauber voneinander abgrenzt.

Das hier vorgestellte Framework bietet diese Möglichkeit. Dabei lässt es offen, wie man eine solche Abgrenzung definieren möchte. Innerhalb einer Unternehmung kann es Rollen geben, deren Sicht auf die Daten nach unterschiedlichen Kriterien in der Breite und Tiefe begrenzt werden müssen, was durch das hier beschriebene Rollenkonzept unterstützt wird.

Abschließend kann festgehalten werden, dass der hier vorgestellte Lösungsansatz wegen seiner Effizienzpotenziale bei CRUD-Operationen und Auswertungen und wegen des integrierten Rollenkonzepts sowohl Frontend- als auch Backendseitig die Entwicklung von Datenbank-Anwendungen in einer serviceorientierten Architektur vereinfacht.

10 Anhang A: Datenselektierende GUI-Komponenten

Nachstehend werden die im Kapitel 3.1 vorgestellten Darstellungstypen datenselektierender GUI-Komponenten ausführlicher beschrieben.

10.1 Tabellen

10.1.1 Zweck

Die tabellarische Darstellung stellt die grundlegendste Form dar, wie Ergebnismengen dem Anwender präsentiert werden können.

Anwendungen nutzen tabellarische Darstellungen zur Anzeige eines Suchergebnisses oder eines Arbeitsvorrats, von dem ausgehend der Anwender weitere Bearbeitungsschritte vornehmen kann – etwa die Bearbeitung eines herausgesuchten Datensatzes.

Dem Anwender wird i. d. R. ein Suchkriterien-Dialog angeboten, in dem er die abgefragten Daten näher spezifizieren kann. Nach dem Auslösen der Suche wird die Ergebnismenge, sofern sie nicht leer ist, tabellarisch dargestellt.

10.1.2 Aufbau

Eine Tabelle in der GUI besteht aus Spalten mit Spaltenüberschriften und darunter angeordneten Daten. Ferner verfügt eine Tabellen-Ansicht über Bedienelemente zur Navigation innerhalb der Daten wie beispielsweise Bildlaufleisten und einen Paginator[18].

10.1.3 Auswertung

Damit das Frontend die gewünschte Darstellung vornehmen kann, enthält eine Abfrage-Definition neben den backendseitig ausgewerteten Angaben, aus denen ein Datenbank-Statement generiert werden kann, noch weitere Angaben für das Frontend – wie z. B. Spaltenüberschriften. Des Weiteren sind Attribute zur Formatierung, Gruppierung und Summierung möglich.

[18] Bei einem Paginator handelt es sich um ein Bedienelement mit Schaltflächen zum seitenweisen Blättern innerhalb einer (tabellarisch dargestellten) Ergebnismenge.

```
32   , { "Name":
33       "CASE WHEN RENTAL.RENTAL_DAY + 21 < IsNull(RENTAL.RETURN_DAY, GetDate())
34             THEN DateDiff(d, RENTAL.RENTAL_DAY + 21, IsNull(RENTAL.RETURN_DAY, GetDate()))
35             ELSE 0
36        END * 0.09",          "Table":"",                "Alias":"PRICE"
37     , "Type":"number",       "formatter":"money",       "bottomCalc":"sum"
38     , "Group":"Ausleihe",    "Label":"Gebühr"
39     }
```

```
68   , { "Table":""
69     , "Name":"MEMBER.FIRST_NAME + ' ' + MEMBER.LAST_NAME + ' - ' + MEMBER.SSN"
70     , "Alias":"FULLNAME"
71     , "Type":"string"
72     , "ShowTable":"no"
73     , "ShowDetail":"no"
74     , "Filter":"no"
75     }
```

```
94 , "TableOptions":
95   { "groupBy":"FULLNAME"
96   , "groupStartOpen":"true"
97   }
```

Definitionsbeispiel 15: Erweiterung der Abfrage-Definition für Ausleihen

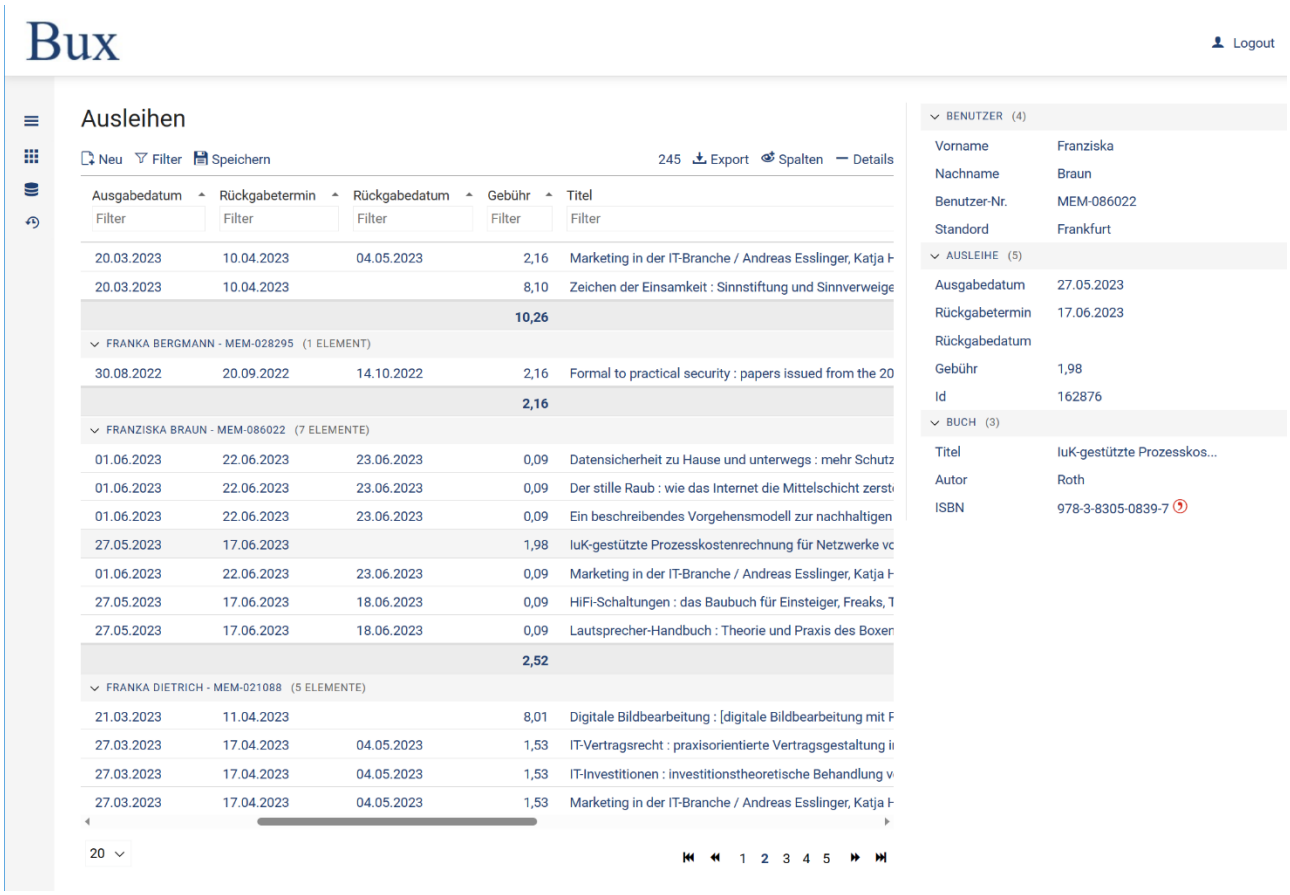
Die erste der drei obigen Erweiterungen an der im Abschnitt *Abfrage-Definitionen* vorgestellten *RENTAL.query* definiert eine weitere Ausgabe-Spalte, in der die Überziehungsgebühr als Betrag formatiert ausgegeben wird (hier wird der Einfachheit halber ein fester Betrag von 9 Ct. pro Tag berechnet. Zudem werden hier datenbankherstellerabhängige Funktionen zum Umgang mit Datums-Ausdrücken verwendet. Die Abstraktion von solchen Abhängigkeiten, sollte in der Praxis vorgenommen werden, ist jedoch nicht Gegenstand dieser Arbeit).

Die zweite Erweiterung setzt in einer weiteren Spalten-Definition einen kompletten Benutzernamen inkl. seiner Benutzernummer zusammen. Dieser Benutzername wird in der dritten Erweiterung, dem Attribut *TableOptions* verwendet, um eine frontendseitig vorgenommene Gruppierung der Ergebniszeilen nach Benutzer vorzunehmen.

Pro Benutzer erscheint in der Tabelle eine Kopfzeile mit seinem vollen Namen inkl. Benutzernummer. Danach folgen seine Ausleihen und dann folgt eine Summenzeile, in der die Spalten aufaddiert werden, deren Spalten-Definition das Attribut *bottomCalc* mit dem Wert *sum* besitzen – also hier die Spalte für die Überziehungsgebühr.

Zusätzlich werden die Spalten mit einer inhaltlichen Gruppe versehen, was zu gruppierten Suchkriterien und zu einer gruppierten Einzelsatzanzeige beim Hovern führt.

Diese Erweiterungen haben auf das generierte SQL-Statement keinen Einfluss, da diese Darstellungsänderungen ausschließlich auf dem Frontend erfolgen.



Screenshot 7: Gruppierte Zeilen und Einzelsatzdarstellung

Weitere Definitionsmöglichkeiten durch zusätzliche Attribute sind denkbar und wurden auch im konkreten Fall von Fleet+ Web umgesetzt, sind jedoch zur Veranschaulichung der hier vorgestellten Konzepte nicht erforderlich und werden daher hier nicht vertieft.

10.1.4 Anforderungen an das Framework:

Zur Umsetzung des hier konzipierten Frameworks sind folgende Anforderungen essentiell:

1. Darstellung sämtlicher Spalten aus der Abfrage-Definition mit der dort als *Label* angegebenen Spaltenüberschrift
2. Wechselmöglichkeit (z. B. über eine Schaltfläche) zwischen Tabellenansicht und Suchkriterien-Ansicht
3. Verwendung der selektierten Spalten als Suchkriterien

Weitere Komfort-Funktionen wie die nachstehend genannten sollten ebenfalls umgesetzt werden. Sie sind nicht spezifisch für Anwendungen, die nach den hier vorgestellten Prinzipien entwickelt werden, sondern auch in anderen Anwendungen anzutreffen. Innerhalb des hier beschriebenen Frontends lassen sich diese Komfort-Funktionen allerdings leicht an zentraler Stelle implementieren und über Definitionen nutzen.

Das Frontend sollte innerhalb einer Abfrage-Definition Möglichkeiten zur Nutzung folgender Komfort-Funktionen bieten. Nachstehend bezieht sich der Begriff „Spalte“ auf ein Element der Columns-Liste innerhalb einer Abfrage-Definition:

1. Angaben von Spalten-Gruppen, nach denen mehrere Spalten in der Einzelsatzdarstellung beim Hovern über einen Datensatz inhaltlich zusammengefasst und gruppiert dargestellt werden
2. Angabe von Gruppierungs-Spalten, nach denen Zeilen mit selbem Wert in diesen Spalten zu Gruppen zusammengefasst werden
3. Angabe, ob eine Spalte pro Gruppe bzw. insgesamt summiert werden soll
4. Angabe, wie die Werte in einer Spalte formatiert werden sollen
5. Angabe, ob im Spaltentitel ein Filter-Feld erscheinen soll
6. Angabe der Schrittweiten im Paginator im Tabellen-Fuß

Weitere Komfort-Funktionen lassen sich ebenfalls zentral im Framework realisieren, bedürfen jedoch keiner speziellen Definitionsmöglichkeit innerhalb von Abfrage-Definitionen:

1. Die Spalten einer Tabelle sollten per Drag&Drop verschiebbar sein. Diese Einstellung sollte pro Anwender gespeichert werden, so dass er die Tabelle beim nächsten Aufruf so vorfindet.
2. Beim Klick auf einen Spaltentitel sollte sich die Anzeige nach der jeweiligen Spalte aufsteigend sortieren, und bei einem erneuten Klick soll sich die Sortierung umkehren.
3. Die Anzahl der in der Tabelle dargestellten Zeilen, sollte ausgegeben werden und sich aktualisieren, wenn über die Filterfelder in den Spaltenköpfen weiter eingeschränkt wird.
4. Es sollte eine Aufrufmöglichkeit (z. B. per Schaltfläche) eines Dialogs geben, mit dem einzelne Spalten ein- und ausgeblendet werden können. Diese Einstellung sollte pro Anwender gespeichert werden, so dass er die Tabelle beim nächsten Aufruf so vorfindet.

5. Eingegebene Suchkriterien sollten unter einem Filter-Namen pro Anwender gespeichert werden können, so dass der Anwender eine Tabelle mit wiederkehrend benötigten Suchkriterien aufrufen kann.
6. Der Inhalt der Tabelle sollte exportiert werden können.
7. Die Einzelsatzdarstellung, die eine Datenzeile beim Hovern rechts neben der Tabelle vertikal darstellt, sollte ein- und ausschaltbar sein.

10.2 Businessgrafiken

10.2.1 Zweck

Businessgrafiken (siehe Komponenten 2a und 2b aus Screenshot 1) bieten einen schnellen Überblick über die Daten und können auch interaktiv gestaltet werden, so dass der Anwender Elemente der Grafik anklicken und damit zu einer vertieften Darstellung gelangen kann.

10.2.2 Aufbau

Dieses Dokument beschränkt sich auf die Beschreibung von Torten- und Balkendiagrammen, da es darum geht, wie man derlei Programmbestandteile definitionsgetrieben realisieren kann. Die hier dargestellten Prinzipien lassen sich jedoch problemlos auf andere Arten von Businessgrafiken übertragen.

10.2.3 Auswertung

Wie in Tabelle 4 beschrieben, wird die Darstellungsform eines Menüpunktes innerhalb der Menü-Definition in seinem Attribut *Type* angegeben. Neben der Darstellungsform *table* für Tabellen kommen nun die Ausprägungen *pie* und *bar* für Torten- und Balkendiagramme hinzu. Für weitere Diagrammtypen sind einfach weitere Ausprägungen und ihre entsprechende Berücksichtigung vorzusehen.

10.2.4 Backendseitige Gruppierung und Spalten-Aliase

Bei datenselektierenden Komponenten, die Daten aggregieren, wie dies häufig bei Businessgrafiken der Fall ist, bietet es sich an, die Daten backendseitig gleich gruppiert zu selektieren, was dadurch realisiert werden kann, dass die Abfrage-Definition im Attribut *Groups* eine Gruppen-Liste enthält.

```
{ "Columns":
  [ { "Table":"LOCATION", "Name":"NAME",     "Alias":"LOCATION"}
  , { "Table":"",         "Name":"COUNT(*)", "Alias":"value"}
  , { "Table":"LOCATION", "Name":"ID",       "Alias":"DRILL_GROUP"}
  ]
, "Tables":
  [ { "Name":"RENTAL"}
  , { "Name":"MEMBER",  "Alias":"MEMBER",   "JoinType":"JOIN"
    , "JoinCondition":"MEMBER.ID   = RENTAL.MEMBER_ID"
    }
  , { "Name":"CATALOG", "Alias":"LOCATION", "JoinType":"JOIN"
    , "JoinCondition":"LOCATION.ID = MEMBER.LOCATION_ID"
    }
  ]
, "Groups":
  [ "LOCATION.NAME"
  , "LOCATION.ID"
  ]
, "Orders":
  [ "2 DESC"
  ]
, "Options":
  [ {"key":"DrillDown",   "value": "RENTAL"}
  , {"key":"DRILL_GROUP", "value": "LOCATION.ID = #<DRILL_GROUP>#"}
  ]
}
```

Definitionsbeispiel 16: Gruppierung in RENTAL_GROUP.query

Durch die Verwendung von Aliasen in den Spalten-Definitionen dieser Abfrage-Definition erhalten die Attribute in den einzelnen Objekten pro Zeile der Ergebnismenge entsprechende Bezeichnungen.

```
[ {"LOCATION":"Essen",      "value":6141, "DRILL_GROUP":735 }
, {"LOCATION":"Berlin",     "value":5342, "DRILL_GROUP":732 }
, {"LOCATION":"Düsseldorf", "value":4837, "DRILL_GROUP":734 }
, {"LOCATION":"Frankfurt",  "value":4711, "DRILL_GROUP":736 }
, {"LOCATION":"Chemnitz",   "value":4605, "DRILL_GROUP":733 }
, {"LOCATION":"Leipzig",    "value":4468, "DRILL_GROUP":741 }
, {"LOCATION":"Köln",       "value":4408, "DRILL_GROUP":740 }
, {"LOCATION":"München",    "value":4376, "DRILL_GROUP":742 }
, ...
, {"LOCATION":"Nürnberg",   "value":2197, "DRILL_GROUP":743 }
, {"LOCATION":"Potsdam",    "value":2082, "DRILL_GROUP":744 }
, {"LOCATION":"Stuttgart",  "value":1463, "DRILL_GROUP":746 }
, {"LOCATION":"Regensburg", "value":1299, "DRILL_GROUP":745 }
]
```

Ergebnismenge 1: Ausleihen pro Standort (Auszug)

10.2.5 Drilldown

Bei Businessgrafiken bietet es sich an, eine Drilldown-Möglichkeit zu schaffen, so dass beim Klicken auf einen Balken eines Balkendiagramms oder auf einen Sektor im Tortendiagramm eine detailliertere, auf den selektierten Ausschnitt beschränkte Ansicht erscheint.

In obigem Fall könnte beispielsweise beim Klicken auf einen Sektor im Tortendiagramm eine Liste der Ausleihen des durch den jeweiligen Sektor repräsentierten Standorts erscheinen.

Das Framework sollte somit über die Möglichkeit verfügen, Drilldowns zu definieren.

Hierzu muss dreierlei definiert werden können:

1. Drilldown-Aktion, die bei solch einem Drilldown gestartet werden soll
2. Drilldown-Bedingung, die die Daten der Drilldown-Aktion einschränkt
3. Drilldown-Kriterium, das in die Drilldown-Bedingung einfließt

In obiger Abfrage-Definition ist dies durch die im Attribut *Options* enthaltenen Key/Value-Paare definiert.

Im key *Drilldown* wird die Drilldown-Aktion in Form der ID des Menüpunkts angegeben, der aufgerufen werden soll – hier *RENTAL*, d. h. der oben bereits vorgestellte Menüpunkt, der eine tabellarische Aufstellung von Ausleihen anzeigt.

Im key *DRILL_GROUP* wird die Drilldown-Bedingung angegeben, die die aufzurufende Ausleihen-Liste auf Ausleihen des betreffenden Standorts einschränkt.

Das Drilldown-Kriterium wird durch die erste Spalte in der *Columns*-Liste der Abfrage-Definition definiert, wenn es nur zwei Spalten in dieser Liste gibt (die zweite definiert den Wert, der pro Gruppe ausgegeben wird). Enthält die Spaltenliste mehrere Spalten, enthält die dritte Spalte das Drilldown-Kriterium. Die erste Spalte definiert dann lediglich die textuelle Darstellung der Gruppen.

Die Spalten-Definition mit dem Drilldown-Kriterium muss ein Attribut namens *Alias* erhalten. Die übrigen können einen *Alias* haben.

Die Drilldown-Bedingung ist jeweils in dem *key* zu finden, der so heißt wie der *Alias* des Drilldown-Kriteriums – hier also *DRILL_GROUP*. Für den Wert des Drilldown-Kriteriums enthält die Drilldown-Bedingung einen Platzhalter, der zur Laufzeit mit dem selektierten Drilldown-Kriterium (sprich mit dem Wert des angeklickten Diagrammelements; hier der Standort-ID) ersetzt wird.

Beim Drilldown wird die Drilldown-Aktion, hier der Menüpunkt *RENTAL*, mit der Datenselektion seiner Abfrage-Definition ausgeführt, die um die Drilldown-Bedingung ergänzt wird.

10.2.6 Vermeidung von SQL-Injection

In Verbindung mit Drilldowns, wie sie bei Businessgrafiken üblich sind, aber auch in anderen Zusammenhängen, ergibt sich das Erfordernis, eine Abfrage um eine zusätzliche Bedingung zu ergänzen.

Die Ermittlung, welcher Teil der Businessgrafik (Balken oder Tortenausschnitt) geklickt wurde – in den Beispielen aus *Screenshot 1*: mit welchem Standort der Drilldown erfolgen soll – geschieht zwar auf Seiten des Frontends, aber zur Vermeidung von SQL-Injection darf die Bedingung auf keinen Fall vom Frontend zum Backend übertragen werden, sondern muss backendseitig ermittelt werden.

Beim Drilldown wird dem Backend bei der Abfrage der Daten zum Einen der Alias und der Wert des Drilldown-Kriteriums mitgegeben und zum Anderen die ID des Menüpunktes, aus dem heraus der Drilldown erfolgte, so dass das Backend wie oben beschrieben aus der Abfrage-Definition dieses Menüpunktes die Bedingung entnehmen und den darin enthaltenen Platzhalter (hier *#<DRILL_GROUP>#*) durch eine Bind-Variable ersetzen kann, die auf den Drilldown-Wert gesetzt wird.

10.2.7 Anforderungen an das Framework

In Verbindung mit Businessgrafiken ergeben sich folgende Anforderungen an das Framework:

1. Das Frontend muss unterschiedliche Arten von Businessgrafiken darstellen können, die im Attribut *Type* des jeweiligen Menüpunktes angegeben werden können müssen
2. Abfrage-Definitionen müssen Angaben zu Drilldowns enthalten können (Drilldown-Aktion, Drilldown-Bedingung, Drilldown-Kriterium)
3. Das Frontend muss beim Klick auf ein Element der Businessgrafik (Balken eines Balkendiagramms, Sektor eines Totendiagramms oder analog bei anderen Grafiktypen) den in der Drilldown-Aktion angegebenen Menüpunkt aufrufen und bei der zugehörigen Selektion der Daten die ID des aufrufenden Menüpunktes und den Wert des Drilldown-Kriteriums an das Backend weitergeben können.
4. Das Backend muss die Drilldown-Bedingung ermitteln können.

5. Das Backend muss das SELECT-Statement für die Drilldown-Aktion generieren und dabei die Drilldown-Bedingung ergänzen können.
6. Das Backend muss das Statement gegenüber der Datenbank absetzen und dabei den Wert des Drilldown-Kriteriums als Bind-Variable mitgeben können.

10.3 Kennzahlen

10.3.1 Zweck

Kennzahlen oder Key Performance Indicators (KPI) bieten einen schnellen Überblick über das Gesamtsystem. Daher ist es sinnvoll, hierfür eine Darstellungsform zu definieren.

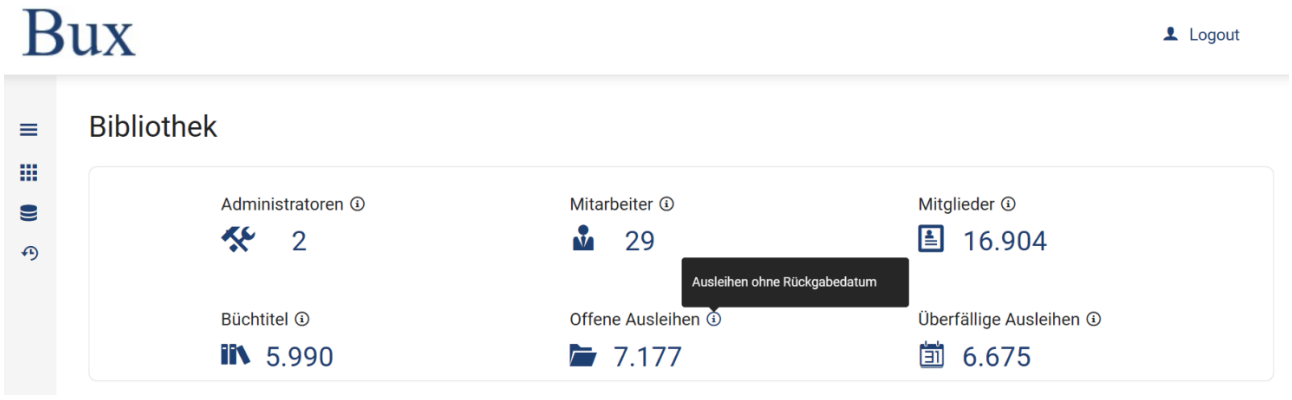
10.3.2 Aufbau

Das Framework sieht hierfür den Type *kpi* vor, der wie die übrigen Darstellungsformen in der jeweiligen Menü-Definition anzugeben ist

Eine Kennzahl besteht in der GUI aus einer Überschrift, einem möglichst aussagekräftigen Icon und einem Wert.

Insbesondere bei KPIs ist es sinnvoll, dem Anwender die Berechnungsgrundlage des KPI transparent zu machen. Daher kann einem KPI auch ein kleines Info-Icon hinzugefügt werden, bei dessen Hovern ein Tooltip mit der Beschreibung der Berechnungsgrundlage angezeigt wird.

Ein solches Info-Icon ist auch bei allen anderen Darstellungsformen denkbar, weshalb es auch bei jedem anderen Menüpunkt angegeben werden können soll.



Screenshot 8: Darstellung einer Kennzahl mit Tooltip

10.3.3 Auswertung

Die Auswertung erfolgt analog zu denen von Tabellen und Businessgrafiken, indem aus der Abfrage-Definition ein SELECT-Statement generiert wird und dessen Daten zum Client gesendet werden. Im Unterschied zu Tabellen und Businessgrafiken besteht die Ergebnismenge allerdings nur aus einem einzigen Wert.

Ein KPI ist eine einzeilige Tabelle mit zwei Spalten, die in der ersten Spalte ein Icon anzeigt und in der zweiten eine Kennzahl. Die Abfrage-Definition hierfür definiert demnach einfach nur COUNT(*) (oder eine andere Aggregat-Funktion bzw. einen einfachen Wert) für die zweite dargestellte Spalte. Für die erste Spalte wird NULL selektiert und als Datentyp „icon“ angegeben. Das Attribut *Icon* gibt an, welches Icon dargestellt wird.

```
{ "Columns":
  [ { "Table":"", "Name": "NULL",     "Type":"icon",   "Icon":"#<Icon>#"}
  , { "Table":"", "Name": "COUNT(*)", "Type":"number", "formatter":"integer"}
  ]
, "Tables":
  [ { "Name":"RENTAL"}
  , { "Name":"MEMBER",  "Alias":"MEMBER",   "JoinType":"JOIN"
    , "JoinCondition":"MEMBER.ID   = RENTAL.MEMBER_ID"
    }
  , { "Name":"CATALOG", "Alias":"LOCATION", "JoinType":"JOIN"
    , "JoinCondition":"LOCATION.ID = MEMBER.LOCATION_ID"
    }
  ]
, "Options":
  [ {"key": "rowClick"
    , "value": "udqUtils.executeWithCurrentFilter(row, 'RENTAL', true)"
    }
  ]
}
```

Definitionsbeispiel 17: KPI-Abfrage-Definition RENTAL_COUNT.query für

Icons werden in Form von CSS-Klassen angegeben, die letztlich einen bestimmten Content aus einer Icon-Font-Sammlung angeben. Für die hier dargestellten Icons wird die Icon-Sammlung IcoMoon [34] verwendet. Andere, für diese Zwecke ebenfalls verwendete Icon-Bibliotheken, wie z. B. Fontawesome [35], funktionieren ähnlich.

```
.icon-folder-open:before {
    content: "\e9dc";
}
.icon-calendar5:before {
    content: "\ea63";
}
```

Definitionsbeispiel 18: CSS für Icon-Klassen aus IcoMoon

Das Icon wird in diesem Fall nicht direkt in der Abfrage-Definition angegeben, was auch möglich wäre, aber, da diese Abfrage-Definition mehrfach verwendet wird (siehe Kapitel *Wiederverwendung von Abfrage-Definitionen*), wird in diesem Fall der Name des zu verwendenden Icons als Parameter aus der Menü-Definition bezogen. Der Platzhalter *#<Icon>#* wird durch den in der Menü-Definition angegebenen Parameter-Wert mit dem Schlüssel *Icon* ersetzt.

Obige Abfrage-Definition für ein KPI besitzt in seinem Options-Attribut ebenfalls ein Drilldown. Da KPIs Tabellen sind, werden Drilldowns hier anders realisiert als Drilldowns aus Businessgrafiken, wo auf das Klicken auf ein Grafik-Element reagiert wird (Balken oder Tortensegment). Abfrage-Definitionen für Tabellen besitzen die Möglichkeit, Event-Handler zu definieren. Im Falle von KPIs kann einfach der Event-Handler für das Klicken auf eine Zeile verwendet werden, um den Aufruf für einen Drilldown zu realisieren.

In obigem Beispiel wird beim Klick auf die Zeile eine JavaScript-Funktion des Frontends des Frameworks aufgerufen, die den im zweiten Parameter angegebenen Menüpunkt ausführt. Der erste Parameter enthält die geklickte Zeile, aus der Informationen über die aufrufende Abfrage ermittelt werden können. Der dritte gibt an, ob der Drilldown in einem separaten Karteireiter erscheinen soll.

Obiges Beispiel ruft also beim Drilldown den im Abschnitt *Abfrage-Definitionen* beschriebenen Menüpunkt mit der ID *RENTAL* auf (Menüpunkt mit dem Label *Ausleihen*). Die verwendete Framework-Funktion sorgt nicht nur für den Aufruf des für den Drilldown angegebenen Menüpunkts, sondern gibt für dessen Datenselektion zusätzlich die Menü-ID des Aufrufers mit, so dass das Backend dessen Filterbedingung hinzuzieht, wenn es die Daten für die Tabelle des Drilldowns liefert.

Wichtig ist nicht nur hier, sondern generell, dass zur Vermeidung von SQL-Injection die Drilldown-Bedingung nicht übers Netz geht, sondern backendseitig an Hand der Menü-ID des Aufrufers ermittelt und gesetzt wird.

Der Aufrufer (also das KPI) gibt seine Menu-ID mit, so dass der Aufgerufene (der Ausleihe-Browser) dem Backend mitteilen kann, dass zusätzlich zu möglichen Filtern in der Abfrage-Definition des Ausleihe-Browsers auch die Bedingung des KPI verwendet wird.

In obigem Beispiel zählt das KPI nur offene Ausleihen – also solche ohne Rückgabedatum. Der Ausleihe-Browser würde ohne diese Bedingung alle Ausleihen anzeigen – also auch solche, die bereits durch Rückgabe des Buches abgeschlossen wurden. Damit der Ausleihe-Browser bei diesem Drilldown jedoch nur diejenigen Ausleihen auflistet, die das KPI zählt, muss backendseitig die Bedingung des Aufrufers an Hand von dessen Menü-ID ermittelt und mit berücksichtigt werden.

10.3.4 Anforderungen an das Framework

KPIs erfordern folgende Definitionsmöglichkeiten:

1. Angabe eines Icons
2. Angabe eines Beschreibungstextes
3. Angabe der Drilldown-Aktion
4. Mitgabe des aufrufenden Menü-Elements, dessen Bedingung beim Drilldown verwendet werden soll.

10.4 Dashboards

10.4.1 Zweck

Dashboards dienen dem Gesamtüberblick, indem sie mehrere Auswertungen gleichzeitig zeigen.

10.4.2 Aufbau

Sie bestehen aus Kennzahlen, Businessgrafiken und/oder Tabellen – siehe *Screenshot 1: Menü und ein Dashboard der Beispielanwendung*.

10.4.3 Auswertung

Dashboards verfügen über keine eigenen Abfrage-Definition, sondern zeigen die unterhalb von ihnen definierten KPIs, Businessgrafiken und/oder Tabellen entsprechend ihren jeweiligen Abfrage-Definitionen an.

Bei Dashboards und den darin enthaltenen Boxen und Karteireitern handelt es sich, wie im Kapitel *Komposition einer Anwendung* beschrieben, um strukturierende Menü-Elemente, die wie das Hauptmenü selbst Untermenü-Elemente für die einzelnen datenselektierenden GUI-Komponenten enthalten.

Oben genanntes Dashboard enthält eine Box mit mehreren KPIs und darunter einen Karteireiter-Container mit zwei Standort-bezogenen Businessgrafiken auf dem ersten Karteireiter und Auswertungen zu Büchern (Bestsellerlisten) auf dem zweiten Karteireiter.

Damit beim Programmstart sofort dieses Dashboard erscheint, erhält dessen Menüpunkt das Attribut *Autostart* mit dem Wert *yes*.

Die KPIs und die Grafiken sollen nebeneinander dargestellt werden, weshalb die Box und der erste Karteireiter jeweils über das Attribut *Columns* eine Spaltenanzahl erhalten und ihre Inhalte eine Angabe zur Spalte besitzen, in der sie angezeigt werden sollen.

10.4.4 Anforderungen an das Framework

Um Dashboards darzustellen werden folgende Definitionsmöglichkeiten benötigt:

1. Gruppierung von Elementen in Boxen und Karteireitern
2. Angabe der Spaltenanzahl
3. Vertikale Darstellung der Spalten bei zu geringer Bildschirmbreite (z. B. auf Smartphones)
4. Darstellung aller untergeordneten Komponenten und deren Unter-Unterkomponenten etc. bei Auswahl eines Menüpunktes des Typs *dash*.
5. Just-in-Time-Selektion der Daten; d. h. Auswertungen auf nicht aktiven Karteireitern oder solche, zu denen man erst runterscrollen muss, bevor sie sichtbar werden, sollten auch erst bei Erscheinen ihre Daten abfragen.

10.5 Einzelsatzdarstellung

10.5.1 Zweck

Neben der Einzelsatzdarstellung, die beim Hovern über eine Tabellenzeile im Detail-Bereich der Tabelle angezeigt wird, werden Einzelsatzdarstellungen auch in Eingabemasken benötigt, wenn zu dem in der Eingabemaske bearbeiteten Objekt ein Fremdschlüsselbezug dargestellt werden soll – etwa dann, wenn z. B. in der Bearbeitungsmaske für eine Buch-Ausleihe Details zum betroffenen Buch oder Benutzer angezeigt werden sollen. Solche Einzelsatzdarstellungen werden in diesem Dokument SDT (**S**ingle Row **D**ata **T**able) genannt (siehe auch Komponenten 3a, b und c in *Screenshot 2* sowie in der dazugehörigen *Tabelle 3*).

In der Eingabemaske bearbeitet wird bei obigem Beispiel ein Datensatz aus der Tabelle *RENTAL*. Diese besitzt jedoch Fremdschlüssel auf die Tabellen *BOOK* und *MEMBER*, um auf die betreffenden Objekte zu verweisen.

Ein solcher Fremdschlüsselbezug soll in der Maske nicht lediglich als Schlüssel angezeigt werden, sondern es sollen einzelne Attribute des bezogenen Objektes (in obigem Beispiel des Buchs und Benutzers) angezeigt werden. Bei der Neuanlage einer Buchausleihe sollen die betroffenen Objekte über die selbe GUI-Komponente ausgewählt werden können.

10.5.2 Aufbau

Ein Bedienelement zur Einzelsatzdarstellung ist im Wesentlichen eine Tabelle, wie sie oben bereits vorgestellt wurden. Entsprechend dem Einsatzzweck als Darstellung von Fremdschlüsselbeziehungen werden für eine solche Tabelle drei Zustände benötigt:

1. Eingabe von Suchkriterien,
2. tabellarische Anzeige der Ergebnismenge
3. Einzelsatzdarstellung eines ausgewählten Eintrags

10.5.3 Auswertung

Realisiert wird dies durch den weiteren Darstellungstyp *sdt*. Erhält ein Menü-Eintrag im Attribut *Type* den Wert *sdt*, so wird an Hand der Abfrage-Definition eine Tabelle generiert, die zwischen diesen drei Modi wechseln kann.

Für die o. g. SDTs können die bestehenden Abfrage-Definitionen verwendet werden. Es wird lediglich jeweils ein zusätzlicher Menü-Eintrag benötigt, der besagt,

dass die betreffenden Abfrage-Definitionen mit jeweils neuer Menü-ID auch als SDT verwendet werden können sollen.

```
{ "Id":"MEMBER_SDT", "Type":"sdt", "Label":"Benutzer", "File":"MEMBER.query"
{ "Id":"BOOK_SDT",   "Type":"sdt", "Label":"Bücher",   "File":"BOOK.query"}
```

Definitionsbeispiel 19: Wiederverwendung einer Query als SDT

In der Abfrage-Definition sind zwei Maßnahmen erforderlich: Zum Einen muss die Primärschlüsselspalte als solche gekennzeichnet werden, damit die ausgewählte Zeile mit dem übergeordneten Objekt verknüpft werden kann. Dies geschieht durch das Attribut *Constraint* und den dortigen Wert *PK*, der in o. g. Beispielen bei der Spalte *ID* gesetzt ist. Und zum Anderen muss es eine Schaltfläche geben, die es ermöglicht, eine Zeile aus der Ergebnistabelle auszuwählen (siehe grünen Haken im Ergebnismodus der Komponente 3a in *Screenshot 2*).

Ansonsten ähnelt eine Abfrage-Definition für SDTs denen von anderen Darstellungsformen.

Für die Auswahlschaltfläche wird in die Spalten-Liste eine Spalten-Definition mit einem entsprechenden Schaltflächen-Typ angelegt, die NULL selektiert. Beim Klicken auf diese Schaltfläche sorgt das Framework dafür, dass die Zeile der betreffenden Zelle selektiert und die Darstellung in den Einzelsatzmodus umschaltet.

```
 1 { "Columns":
 2   [ { "Button":"select",     "Name":"NULL"
 3     , "Condition":"QueryArea.getType() == 'sdt'"
 4     }
…
24   , { "Table":"BOOK",        "Name":"ID",                "Alias":"ID"
25     , "Type":"number",       "Label":"Id", "Constraint":"PK", "Serial":"AUTO"
26     }
27   ]
28 , "Tables":
29   [ { "Name":"BOOK"}
30   ]
```

Definitionsbeispiel 20: Abfrage-Definition BOOK.query für ein SDT

10.5.4 Anforderung an das Framework

Folgendes muss in einer Abfrage-Definition angegeben werden können, um SDTs zu realisieren:

1. Kennzeichnung der PK-Spalte
2. Spalte mit Schaltflächen-Typ zur Datensatzauswahl
3. Umschaltung zwischen den drei Modi

10.6 Darstellung von untergeordneten Objekten

10.6.1 Zweck

Die vollständige Abbildung von Master-Detail- oder 1:n-Beziehungen erfordert die Darstellbarkeit beider Blickrichtungen: Ist man in der Darstellung eines Detail-Objekts, benötigt man ggf. zusätzlich die Darstellung von Attributen des Masters, was wie oben beschrieben mit SDTs möglich ist (Beispiel: Buch einer Ausleihe).

Umgekehrt kann es der Fall sein, dass man gemeinsam mit der Darstellung eines Masters zusätzlich die von ihm abhängigen Daten tabellarisch darstellen möchte.

10.6.2 Aufbau

Eine Master-Detail-Beziehung aus Sicht des Masters besteht aus einer führenden Komponente – z. B. einem SDT für den Master – und einer oder mehreren davon abhängigen Komponenten – SDTs, KPIs, Businessgrafiken oder Tabellen.

Zur Veranschaulichung einer solchen Master-Detail-Beziehung aus Sicht des Masters verfügt die Beispiel-Anwendung über ein Dashboard *Ein Benutzer*, das als Schaltzentrale für die weitere Arbeit mit dem dort im oberen Teil ausgewählten Bibliotheks-Benutzer fungiert (siehe Hintergrund von *Screenshot 2*). Im unteren Teil sieht man die Ausleihen dieses Benutzers.

10.6.3 Auswertung

Damit Master und Detail zusammenwirken (sprich: eine Aktualisierung im unteren Teil erfolgt, wenn oben ein anderer Benutzer ausgewählt wird), bekommt der oben dargestellte Master (das SDT für die Auswahl eines Benutzers) ein Attribut namens *Dependent* mit dem Wert *yes*, um zu signalisieren, dass andere Menüpunkte von ihm abhängig sind. Diese haben ein Attribut *Master*, in dem ihr Master – nämlich o. g. SDT – an Hand seiner Menü-ID angegeben ist.

Gibt es nun eine Datensatzänderung im Master, so kann das Framework alle davon abhängigen GUI-Komponenten aktualisieren. Für die Aktualisierung wird der Primärschlüssel des Masters an die abhängigen Abfragen weitergereicht. Diese haben eine Spalten-Definition, die damit korrespondiert und durch das Attribut *Constraint* mit dem Wert *ParentPK* als solche gekennzeichnet sind.

Die im Kapitel *Abfrage-Definitionen* vorgestellte Abfrage-Definition *RENTAL.query* wird also um eine Spalten-Definition für den übergeordneten Schlüssel ergänzt, so dass sie nicht nur als Ausleihe-Browser aus dem Menü heraus aufgerufen

werden kann, sondern auch als von einem SDT abhängige Tabelle im Dashboard *Ein Benutzer*.

```
55  , { "Table":"RENTAL",      "Name":"MEMBER_ID",        "Alias":"MEMBER_ID"
56    , "Type":"number",       "Label":"MEMBER_ID"
57    , "ShowTable":"no"
58    , "ShowDetail":"no"
59    , "Filter":"no"
60    , "Constraint":"ParentPK"
61    }
```

Definitionsbeispiel 21: übergeordneter Schlüssel in abhängiger Definition

In diesem Fall ist also die Spalte *MEMBER_ID* als diejenige Spalte gekennzeichnet, die den PK der übergeordneten Abfrage entgegennimmt. Wenn die selbe Abfrage direkt aus dem Menü (*Vorgänge/Ausleihen*) aufgerufen wird, erhält sie keinen Wert für den ParentPK. Dieser Constraint würde dabei ignoriert werden und man kann (entsprechend seinen Rollenfiltern) sämtliche Ausleihen des Systems sehen, während man aus dem Dashboard *Ein Benutzer* nur diejenigen Ausleihen sieht, die zu dem als Master ausgewählten Benutzer gehören.

10.6.4 Anforderungen an das Framework

Zur Darstellung untergeordneter Objekte bedarf es folgender Definitionsmöglichkeiten:

1. Angabe, ob von einem Menüpunkt andere Menüpunkte abhängig sind (`"Dependent":"yes"`)
2. Angabe, von welchem anderen Menüpunkt ein Menüpunkt abhängig ist (`"Master":"Id des Masters"`)
3. Angabe der Schlüssel-Spalte(n), über die die Abhängigkeit definiert ist

Für Punkt 1 kann theoretisch auf eine explizite Definition verzichtet werden, da die Tatsache, dass von einem Menüpunkt andere Menüpunkte abhängig sind, aus deren Attribut *Master* abgeleitet werden kann. Um jedoch nicht bei jeder Datensatz-Änderung jeder Komponente ermitteln zu müssen, ob andere Komponenten davon abhängig sind, sollte ein Master-Menüpunkt das Attribut *Dependent* besitzen (entweder, wie hier gezeigt, explizit oder beim Aufbau seiner abhängigen Komponenten dynamisch gesetzt).

10.7 Kombinationslistenfelder

10.7.1 Zweck

Kombinationslistenfelder, auch Comboboxen oder Dropdowns genannt, dienen der Auswahl eines Datensatzes aus einer (i. d. R. einspaltigen) aufklappbaren Liste von Datensätzen. Ihr Einsatzgebiet ähnelt dem von SDTs.

Üblicherweise werden einfache Kataloge (z. B. die Auswahl von Standorten oder Rollen) als Kombinationslistenfeld dargestellt, während Auswahlen, bei denen mehrere Attribute pro Datensatz angezeigt werden sollen oder wo auf Grund der Anzahl der zur Auswahl stehenden Datensätze der Auflistung eine Eingabemöglichkeit für Suchkriterien vorgeschaltet werden soll, praktischerweise mittels SDTs umgesetzt werden.

10.7.2 Aufbau

Ein Kombinationslistenfeld besteht aus einem Eingabefeld mit einem Label und einer Schaltfläche zum Aufklappen der Liste mit den zur Verfügung stehenden Werten – siehe Element 6 aus *Screenshot 2*.

10.7.3 Auswertung

Die Auswertung erfolgt wie bei allen übrigen hier vorgestellten Darstellungsformen von Komponenten auch, indem aus einer Abfrage-Definition ein SELECT-Statement generiert wird.

Der Wert der jeweiligen Komponente entspricht dem Wert der in der Abfrage-Definition als PK gekennzeichneten Spalte aus der selektierten Zeile.

10.7.4 Anforderungen an das Framework

Für Kombinationslistenfelder sind lediglich folgende Definitionsmöglichkeiten erforderlich:

1. Für das Attribut *Type* wird die zusätzliche Ausprägung *dropdown* benötigt.
2. Wenn die verwendete Abfrage-Definition mehrere Spalten selektiert als in der Dropdown-Liste angezeigt werden sollen, muss gekennzeichnet werden können, welche tatsächlich in die Liste aufgenommen werden sollen.

10.8 Abschließende Bemerkungen zu Darstellungsformen

Dieses Kapitel beschreibt einige Arten von datenselektierenden GUI-Komponenten, die im Kern alle ähnlich definiert werden, indem ihre Datenselektion in einer Abfrage-Definition und ihre Darstellungsform in einem Eintrag einer Menü-Definition festgelegt wird.

Innerhalb dieses Prinzips kann die Anzahl der Darstellungsformen nach Belieben ergänzt werden.

11 Anhang B: Exkurse

11.1 Plausibilisierung einer Menü-Definition

Die Attribute *Id* und *Type* sollten als Pflichtangaben angesehen werden und Menüpunkte ohne diese Angaben sollten vom Framework als ungültig erkannt und samt ihren Unterpunkten aus der Struktur entfernt werden.

Je nach Attribut *Type* kann *File* zur Pflichtangabe werden, so dass dem widersprechende Menüpunkte ebenfalls als ungültig vom Framework aus der Menüstruktur entfernt werden sollten.

Es dürfen nur gültige Menü-Definition ausgeliefert werden. Daher wird empfohlen, ein Prüfprogramm zu erstellen, das eine Menü-Definition im Zuge des Komponenten-Tests auf Gültigkeit prüft. Neben obigen Regeln zur Behandlung fehlender Angaben gehört die Syntax-Prüfung hinsichtlich eines gültigen JSON-Formats dazu, wobei JSON Lines, also JSON mit erlaubten Zeilenumbrüchen innerhalb der Werte, empfohlen wird.

11.2 Alternative Formate

Das Vorhandensein einer ID könnte syntaktisch erzwungen werden, wenn eine Menüebene nicht als Array abgebildet würde, sondern wie in nachstehendem Beispiel als Objekt, wobei die IDs der Menüpunkte jeweils die Attribute des Objekts wären. Dadurch wäre sichergestellt, dass jeder Menüpunkt eine ID hat und dass sich diese IDs innerhalb eines Objekts unterscheiden.

```
{ "STAMMDATEN":
  { "Type":"menu",  "Label":"Stammdaten"
  , "Icon":"icon-database",    "_children":
    { "MEMBER":
      { "Type":"table", "Label":"Benutzer"
      , "File":"MEMBER.query", "CRUD":"CRUD"
      , "Parameters":
        [ { "key":"Alias",     "value":"MEMBER"}
        ]
      }
    , "BOOK":
      { "Type":"table", "Label":"Bücher"
        , "File":"BOOK.query",   "CRUD":"CRUD"
      }
    }
  }
, "VORGANG":
  { "Type":"menu",  "Label":"Vorgänge"
  , "Icon":"icon-history",     "_children":
    { "RENTAL":
      { "Id":,         "Type":"table", "Label":"Ausleihen"
      , "File":"RENTAL.query", "CRUD":"CRUD"
      }
    }
  }
}
```

Definitionsbeispiel 22: Alternative Notation mit erzwungenen IDs

Da die Eindeutigkeit auf diese Weise jedoch ausschließlich pro Objekt und nicht über das komplette Menü sichergestellt werden kann, böte diese Notation letztlich nur den Vorteil irgendeiner vorhandenen ID und macht das Erfordernis der mittels eines wie oben beschriebenen Prüfprogramms automatisch durchführbaren Gültigkeitsprüfung nicht entbehrlich. Zudem ist die ID inhaltlich als Attribut des jeweiligen Menüpunktes anzusehen, weshalb empfohlen wird, sie auch innerhalb des jeweiligen Menüpunkt-Objekts und nicht bei dessen übergeordnetem Objekt zu notieren.

Zudem kann auch eine alternative Dateinamenskonvention verwendet werden, bei der die letztliche Dateinamenserweiterung dem tatsächlichen Dateiformat entspricht (also .jsonl) und zur Erkennbarkeit des inhaltlichen Definitionstyps

noch ein weiterer Dateinamensteil angehängt wird (hier .menu) – also z. B.: ADMIN.menu.jsonl.

Diese Dateinamenskonvention hätte den Vorteil, dass handelsübliche Editoren ohne weitere Einstellungen das Dateiformat sofort erkennen und eine Syntaxhervorhebung vornehmen können.

Diese Arbeit legt keine konkrete Dateinamenskonvention fest, geht jedoch davon aus, dass eine konkrete Dateinamenskonvention verwendet wird.

Ähnliches gilt für die Formatierung. Es wird als hilfreich angesehen, kleine Objekte (hier einzelne Untermenüpunkte) als Einzeiler zu schreiben und große Objekte (hier Hauptmenüpunkte) mehrzeilig zu schreiben, wobei die schließende Klammer genau unterhalb der öffnenden Klammer positioniert wird und Kommas zwischen den Elementen eines solchen Objektes ebenfalls auf der selben Fluchtlinie wie diese beiden Klammern stehen. Doch auch dies soll mit diesem Konzept nicht vorgegeben werden.

Ferner werden in den hier vorgestellten Definitionsbeispielen des Öfteren Arrays mit Key/Value-Paaren verwendet – z. B. auch in den Menü-Parametern der Zeilen 7 bis 9 des vorgenannten Beispiels. Auch diese ließen sich als Dictionary ablegen statt als Array, was doppelte Bezeichner verhindern würde.

11.3 Rollen-Filter vs. Row-Level-Security

Die mit Rollen-Filtern realisierte Anforderung ist inhaltlich zwar vergleichbar mit Row-Level-Security von Microsoft SQL Server, aber da das Framework möglichst unabhängig vom Datenbankhersteller sein soll und Row-Level-Security eine für Web-Anwendungen unpraktikable anwenderspezifische Datenbankverbindung erfordert, damit der SQL Server auch unterscheiden kann, welche Berechtigungen vorliegen, wird obige Anforderung vom Framework nicht mittels Row-Level-Security, sondern definitionsgetrieben realisiert.

Zudem kann eine datenbankserverseitige Row-Level-Security zwar ermitteln, auf welche Zeilen einer Tabelle ein Anwender zugreifen darf, aber dies kann nicht kontextabhängig variieren, da der Datenbankserver den inhaltlichen Kontext nicht kennt. Beispielsweise darf ein Anwender mit der Rolle *MEMBER*, der nur seine eigenen Benutzerdaten sehen darf, in bestimmten Kontexten auch Benutzerdaten anderer Benutzer sehen – nämlich die der Mitarbeiter seines Standorts, die ihm als Ansprechpartner angezeigt werden.

11.4 Auswahl von zu filternden Tabellen

In der Betrachtung eines Gesamt-Statements kann es zu wirkungslosen Filtern kommen, wenn beispielsweise ein Filter eine Tabelle bereits so einschränkt, dass der Filter für eine weitere Tabelle nicht weiter einschränkt. Doch eine inhaltliche Überschneidung der Filter-Definitionen bzw. eine Optimierung des letztendlichen SELECT-Statements kann dem Query-Optimizer des Datenbankmanagementsystems überlassen werden.

Wichtig ist, dass man pro Tabelle festlegen kann, welchen Ausschnitt daraus Anwender einer bestimmten Rolle sehen dürfen, und dass bei jedem aus einer Abfrage-Definition heraus generierten Statement für alle darin enthaltenen Tabellen jeweils ein einschränkendes Sub-Select hinzugefügt wird, sofern es für die Kombination aus Tabelle und Rolle eine Filter-Definition gibt.

In der Praxis muss auf vergleichsweise wenige Tabellen ein Filter gelegt werden, da diese Tabellen in den meisten Kontexten betroffen sind.

Ein Filter auf die Tabelle *RENTAL* erübrigt sich beispielsweise, da Ausleihen kaum ohne Einbindung des betroffenen Benutzers selektiert werden – sprich: Eine Abfrage-Definition für die Tabelle *RENTAL* kann die Tabelle *MEMBER* in der Tabellen-Liste enthalten, so dass der in Abschnitt *Filter-Definitionen* vorgestellte MEMBER-Filter greift. Da der Fremdschlüssel *MEMBER_ID* in der Tabelle *RENTAL* ein Pflichtfeld ist, kann die Tabelle *MEMBER* immer per INNER JOIN an die Tabelle *RENTAL* geknüpft werden, ohne die Ergebnismenge einzuschränken. Die Tabelle

MEMBER kann also auch in solche Ausleihe-Abfragen aufgenommen werden, wo keine Benutzer-Attribute selektiert werden sollen. Die Aufnahme in die Abfrage-Definition würde automatisch zur entsprechenden Filterung der Ausleihen nach den Sichtbarkeitsregeln des Anwenders führen, so dass eine eigene Filter-Definition für die Tabelle *RENTAL* entbehrlich wird. Es müsste allerdings im Entwicklungs-Team Konsens darüber bestehen, dass gewisse Tabellen immer mit aufgenommen werden sollen, um eine entsprechende Filterung sicherzustellen. Das Vorhandensein mindestens einer filterbaren Tabelle in jeder Abfrage-Definition ist ein sinnvoller Testfall für ein hier nicht weiter betrachtetes Testkonzept.

Obiges Beispiel zeigt, dass es sinnvoll erscheint, die Existenz überschneidender Filter-Definitionen zuzulassen, da es auf diese Weise reicht, einer Abfrage-Definition eine einzige mit Filter versehene Tabelle anzufügen – und zwar die am direktesten verknüpfte.

Das sehr einfache Datenmodell der in diesem Dokument beschriebenen Bibliotheksanwendung gibt kaum ein sinnvolles Beispiel für diesen Aspekt her, weshalb auf ein Beispiel aus der Fuhrparkmanagementsoftware Fleet+ Web zurückgegriffen wird:

Durch die Existenz eines Filters auf die Tabelle *FAHRZEUG* reicht es, einer Abfrage für die Tabelle *SCHADENS_BERICHT* die Tabelle *FAHRZEUG* anzufügen, da sich jeder Schadensbericht auf ein Fahrzeug bezieht und durch diese Verknüpfung sichergestellt ist, dass ein Anwender nur Schadensberichte sehen kann, deren Fahrzeuge er auch sehen darf.

Gäbe es keinen Filter für Fahrzeuge, müsste einer Abfrage zu Schadensberichten zusätzlich zur Tabelle *FAHRZEUG* noch die Tabelle *G_PARTNER* angefügt werden, um über den Geschäftspartner (sprich Halter) des beschädigten Fahrzeugs zu filtern, denn letztlich ist die Frage, welche Fahrzeuge ein Anwender sehen darf, identisch mit der Frage, wessen Fahrzeuge er sehen darf, was über den Halter geregelt wird. Schadensberichte über den Halter zu filtern, ist somit direkter als dies über das Fahrzeug zu tun, was letztlich über dessen Halter geht.

Die Anzahl der Tabellen, für die es den o. g. Konsens geben sollte, erreicht beispielsweise in Fleet+ Web nicht einmal das Dutzend, obwohl das Datenmodell über 500 Entitätstypen aufweist. Grund für die geringe Anzahl von Tabellen, für die man einen Filter definieren muss, ist, dass sich die meisten Geschäftsprozesse im Kern um wenige Entitätstypen drehen – es handelt sich halt um ein Fuhrparkmanagementsystem. Da geht es in erster Linie um Fahrzeuge und Fahrer und davon abhängig um Verträge, Schäden, Ein- und Ausgangs-Rechnungen u. s. w. Doch die Anzahl der Entitätstypen im Zentrum der Betrachtung ist überschaubar.

11.5 Paginierung

Die in Tabellen dargestellten Ergebnismengen können unübersichtlich groß werden. Daher verfügen Anwendungen mit tabellarischen Ergebnismengen häufig über einen Paginator im Kopf und/oder Fuß der Tabelle.

Man kann frontendseitiges und backendseitiges Paginieren unterscheiden. Beim frontendseitigen Paginieren wird lediglich der in der Tabelle dargestellte Ausschnitt der Ergebnismenge eingeschränkt und dem Benutzer die Möglichkeit gegeben, mit Videorekorderknöpfen (Schaltflächen, deren Symbole ältere Anwender an die Symbole auf den Vor- und Rückspul-Tasten von Videorekordern erinnern) weitere Seiten aufzurufen. Diese Möglichkeit sollte vom Frontend unterstützt werden.

Beim backendseitigen Paginieren wird der vom Backend gelieferte Teil der Ergebnismenge jeweils auf die angeforderte Seite eingeschränkt. Dies reduziert den Netzwerkverkehr auf die jeweils angeforderte Seite und erspart das Übertragen von Datensätzen, die am Ende gar nicht erst betrachtet werden.

Letzteres ist bei Suchergebnissen mit absteigender Relevanz ein probates Mittel, zügig den relevantesten Teil der Ergebnismenge anzuzeigen und weniger relevantes nur bei Bedarf (oder nebenläufig) nachzuladen.

Um frontendseitige Komfort-Funktionen wie eine Sortierung oder Filterung über die Spaltentitel zu ermöglichen, ist es jedoch erforderlich, dass dem Frontend die gesamte Ergebnismenge vorliegt bzw. just in time nachgeladen wird. Daher ist in der jeweiligen Anwendung fachlich zu klären, ob und welche Suchkriterien durch den Benutzer eingegeben werden müssen, so dass von vorn herein eine handhabbare Ergebnismenge entsteht und auf backendseitiges Paginieren verzichtet werden kann.

Sollte man sich dennoch für backendseitiges Paginieren entscheiden, ist zu bedenken, dass bei einer Web-Anwendung von einem zustandslosen System auszugehen ist, was bedeutet, dass ein am Backend eingehender Request ohne entsprechenden Managementaufwand zunächst einmal nichts von vorangegangenen Requests weiß, die sich auf die selbe Abfrage beziehen. Die Abfrage der n-ten Seite einer zuvor abgesetzten Recherche sollte keinesfalls zur erneuten Datenbankabfrage führen, sondern lediglich weitere (und somit backendseitig zwischenzuspeichernde) Daten der ursprünglichen Abfrage liefern, um erstens den Datenbankserver nicht durch Mehrfachabfragen zu belasten und zweitens die Lesekonsistenz zu gewährleisten – sprich: den Datenzustand zum Abfragezeitpunkt abzubilden.

12 Anhang C: Verzeichnisse

12.1 Abbildungsverzeichnis

12.2 Tabellenverzeichnis

12.3 Definitionsbeispiele

12.4 Verzeichnis generierter Ausgaben

12.5 Quellcodebeispiele

12.6 Literaturverzeichnis

[1] Stackoverflow: What is boilerplate code?. Online verfügbar unter https://stackoverflow.com/questions/3992199/what-is-boilerplate-code, zuletzt abgerufen am 22.08.2023.

[2] Späth, Peter (2021): Beginning Java MVC 1.0. Berkeley, CA: Apress.

[3] Java Specification Requests 371: JSR 371: Model-View-Controller (MVC 1.0) Specification. Online verfügbar unter https://jcp.org/en/jsr/detail?id=371, zuletzt abgerufen am 10.07.2023.

[4] Visual Paradigm: Visual Paradigm User's Guide, Part IX. Code engeneerint. Online verfügbar unter https://www.visual-paradigm.com/support/documents/vpuserguide/276/381/7486_generateorup.html, zuletzt abgerufen am 10.07.2023.

[5] MD Consulting: Nachlese Gupta DevDay 05.11.2019 München. Online verfügbar unter https://www.md-consulting.de/kategorie/unternehmen/page/4/, zuletzt abgerufen am 10.07.2023.

[6] Tiangolo: FastAPI Tutorial - User Guide. Online verfügbar unter https://fastapi.tiangolo.com/tutorial/dependencies/, zuletzt abgerufen am 10.07.2023.

[7] Mathes, Prof. Markus A.; Seufert, Prof. Jochen (2022): Programmieren in C++ für Elektrotechniker und Mechatroniker. Wiesbaden: Springer Fachmedien Wiesbaden.

[8] Yung, Simon Yun Pui (1992) Definitive programming : a paradigm for exploratory programming. PhD thesis, University of Warwick.

[9] Raymond, Eric S. (2003): The Art of Unix Programming: Addison-Wesley Professional.

[10] Robbins, Arnold (2015): Effective awk programming. Universal text processing and pattern matching. 4th edition (Online-Ausg.). Sebastopol, CA: O'Reilly Media.

[11] Brian W. Kernighan, Die UNIX-Story (2020), dpunkt.verlag, Heidelberg, ISBN: 9783969100721

[12] Barua, Anurag (2013): First Steps in SAP Crystal Reports for Business Users: Espresso Tutorials.

[13] Silvério, Diogo Rafael Rebocho, Artur Dias, Mário Pereira, 2022. Efficient Declarative Programming in OCaml

[14] Low-Code Association e.V.: Mitglieder. Online verfügbar unter https://www.lowcodeassociation.org/mitglieder/, zuletzt abgerufen am 10.07.2023.

[15] Low-Code Association e.V.: Das Low-Code Manifest. Online verfügbar unter https://www.lowcodeassociation.org/manifest/, zuletzt abgerufen am 10.07.2023.

[16] Forrester: New Development Platforms Emerge For Customer-Facing Applications, zitiert in Scopeland: Sieben Fakten zur Schlüsselrolle von Low-Code-Plattformen bei der Digitalen Transformation. Online verfügbar unter https://www.scopeland.de/fakten-low-code, zuletzt abgerufen am 10.07.2023.

[17] Bock, Alexander C.; Frank, Ulrich (2021): Low-Code Platform. In: Business & Information Systems Engineering 63 (6), S. 733–740. DOI: 10.1007/s12599-021-00726-8.

[18] Gartner, Magic Quadrant for Enterprise Low-Code Application Platforms (Paul Vincent, Kimihiko Iijima, Adrian Leow, Mike West, Oleksandr Matvitskyy), Januar 2023

[19] Mendinx (2021): The State of Low-Code 2021. Online verfügbar unter https://mendix.com/, zuletzt aufgerufen am 19.06.2023.

[20] PC Magazine, „Google App Maker“. 12.07.2017, Online verfügbar unter https://uk.pcmag.com/cloud-services/89780/google-app-maker, zuletzt aufgerufen am 07.09.2023

[21] Google, „Google Workspace“. 27.01.22020, Online verfügbar unter https://workspaceupdates.googleblog.com/2020/01/app-maker-update.html?m=1, zuletzt aufgerufen am 07.09.2023

[22] J. Varajão, "Software development in disruptive times", Commun. ACM, vol. 64, no. 10, pp. 32-35, 2021.

[23] A. Trigo, J. Varajão and M. Almeida, "Low-Code Versus Code-Based Software Development: Which Wins the Productivity Game?," in IT Professional, vol. 24, no. 5, pp. 61-68, 1 Sept.-Oct. 2022, doi: 10.1109/MITP.2022.3189880.

[24] Binzer, B., Winkler, T.J. (2022). Democratizing Software Development: A Systematic Multivocal Literature Review and Research Agenda on Citizen Development. In: Carroll, N., Nguyen-Duc, A., Wang, X., Stray, V. (eds) Software Business. ICSOB 2022. Lecture Notes in Business Information Processing, vol 463. Springer, Cham.

[25] Betty blocks: The Betty Blocks Guide to Citizen Development. Online verfügbar unter https://www.bettyblocks.com/citizen-developer, zuletzt abgerufen am 10.07.2023.

[26] Cabot, Jordi (2020): Positioning of the Low-Code Movement within the Field of Model-Driven Engineering. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. New York, NY, USA: Association for Computing Machinery (MODELS ’20).

[27] Oli Folkerd: Tabulator. Online verfügbar unter https://tabulator.info/, zuletzt abgerufen am 10.07.2023.

[28] Microsoft: SQL Server 2022, Row-Level Security. Online verfügbar unter https://learn.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver16, zuletzt abgerufen am 10.07.2023.

[29] jQuery. Online verfügbar unter https://jquery.com/, zuletzt abgerufen am 10.07.2023.

[30] jQuery Validate. Online verfügbar unter https://jqueryvalidation.org/, zuletzt abgerufen am 05.08.2023.

[31] Microsoft, „Access“. Online verfügbar unter https://www.microsoft.com/de-de/microsoft-365/access, zuletzt abgerufen am 07.09.2023.

[32] Microsoft, „Verknüpfen von Tabellen und Abfragen“. Online verfügbar unter https://support.microsoft.com/de-de/office/verkn%C3%BCpfen-von-tabellen-und-abfragen-3f5838bd-24a0-4832-9bc1-07061a1478f6, zuletzt abgerufen am 07.09.2023.

[33] SAP, „Benutzerhandbuch für SAP Crystal Reports 2013“. Online verfügbar unter https://help.sap.com/doc/businessobject_product_guides_cr2013_de_cr13sp6_cr_usergde_de_pdf/2013.6/de-DE/cr13sp6_cr_usergde_de.pdf#page=75, zuletzt abgerufen am 04.10.2023.

[34] IcoMoon. Online verfügbar unter https://icomoon.io/, zuletzt abgerufen am 10.07.2023.

[35] Font Awesome. Online verfügbar https://fontawesome.com/, zuletzt abgerufen am 10.07.2023.