Eine datenzentrierte Entwicklungsumgebung für den Schulunterricht

Master-Thesis im Fachbereich Informatik

Eingereicht am: 31. Oktober 2016

Autor: Marcus Riemer, B.Sc.
Matr.-Nr.: 100478
E-Mail: mri@fh-wedel.de

Betreuer:	PD. Dr. Frank Huch	Prof. Dr. Ulrich Hoffmann
E-Mail:	fhu@informatik.uni-kiel.de	uh@fh-wedel.de

Abstract

Konventionelle Entwicklungsumgebungen sind speziell auf die Bedürfnisse von professionellen Anwendern zugeschnittene Programme. Aufgrund der damit verbundenen Komplexität sind sie aus didaktischer Sicht nicht für die Einführung in die Programmierung geeignet. Diese Thesis beschreibt daher ein Konzept und die prototypische Implementierung einer Lehr-Entwicklungsumgebung für Datenbanken und Webseiten namens BlattWerkzeug. Um syntaktische Fehler während der Programmierung systematisch auszuschließen, werden die Bestandteile der dafür benötigten Programmier- oder Textauszeichnungssprachen ähnlich wie in der Lehrsoftware „Scratch“ grafisch durch Blockstrukturen repräsentiert. Diese Blöcke lassen sich über Drag & Drop-Operationen miteinander kombinieren, die syntaktischen Strukturen von `SQL` und `HTML` sind für Lernende dabei stets sichtbar, müssen aber noch nicht verinnerlicht werden. So lassen sich auch ohne die manuelle Eingabe von Codezeilen eigene Webseiten programmieren, welche dann im Freundes- und Bekanntenkreis weitergegeben werden können. Für den Unterrichtseinsatz ist der aktuelle Entwicklungsstand von BlattWerkzeug allerdings noch nicht geeignet, er dient vornehmlich der Erprobung und Demonstration der erdachten Konzepte.

Conventional development environments are programs that are tailored to suit the needs of professionals. Due to their complexity they do not lend themselves well to introduce pupils to programming. This thesis therefore describes the concept and the prototypical implementation of an educational software for database- and web-development named BlattWerkzeug (a loose translation of the english term “Page Tool”). To eliminate the possibility of syntactical errors while programming, the elements of the programming- or markup-languages are represented by graphical blocks, similar to the approach taken by the software “Scratch”. These blocks can be combined by using drag & drop operations. The syntactical structures of `SQL` and `HTML` are not hidden from the user, but it is not mandatory to internalize them. This approach allows pupils to program and share their own websites, without the need to type lines of code. The current implementation of this software is not yet ready to be used in a classroom. It’s main purpose is to demonstrate and field test the described concepts.

Inhaltsverzeichnis

1. Einleitung **1**

2. Vergleichbare Arbeiten und Produkte **6**

2.1. Software: Scratch 6
2.2. Software: App Inventor 8
2.3. Kurs: SqlZoo 10
2.4. Software: Jekyll 11
2.5. Software: Visual Studio Lightswitch 15
2.6. Buch: Die Macht der Abstraktion 17
2.7. Software: MySQL Workbench 18

3. Anforderungsanalyse **21**

3.1. Zielgruppe 22
3.2. Grundprinzipien 23
3.3. Künftigen Erweiterungen vorbehalten 26
3.4. Allgemeines Konzept der Entwicklungsumgebung 27
3.4.1. Webanwendung 28
3.4.2. Projektbasiert 30
3.4.3. Zugriffskontrollen 31
3.4.4. Unterschiedliche Versionen eines Editors 33
3.5. Konzept für sinnvolle Teilmengen von `SQL` 34
3.5.1. Mögliche Einschränkungen der Komponenten 35
3.5.2. Mögliche Einschränkungen der Ausdrücke 38
3.5.3. Sprachstufen 41
3.6. Drag & Drop-Editor für `SQL` 43
3.6.1. Visuelle Gestaltung 44
3.6.2. Grundsätzlicher Aufbau 45
3.6.3. Umgang mit Ausdrücken 48
3.6.4. Abfragen mit Parametern 49
3.6.5. `FROM` 51
3.6.6. `WHERE` 51
3.6.7. `GROUP BY` 52
3.6.8. Manipulation von Daten 52

3.7. Konzept für Oberflächen 54
3.7.1. Datenquellen für Webseiten 56
3.7.2. Evaluation existierender Templatingsprachen 57
3.7.3. Die Templatingsprache Liquid 66
3.7.4. Layout 69
3.7.5. HTML-Bedienelemente 70
3.7.6. Komplexe Bedienelemente 72
3.7.7. Bindung von Abfragen an die Oberfläche 73
3.7.8. Eingabe von Daten über die Oberfläche 76
3.7.9. Navigation 77
3.8. Drag & Drop-Editor für Oberflächen 79
3.8.1. Vor- und Nachteile von WYSIWYG-Darstellungen 80
3.8.2. Vor- und Nachteile einer Blockdarstellung 82
3.8.3. Mögliche Ansätze 84
3.8.4. Grundsätzlicher Aufbau 87
3.8.5. Seitenleiste: Editieren von Eigenschaften 89
3.8.6. Seitenleiste: Angebot von Bedienelementen 90
3.8.7. Seitenleiste: Angebot von verfügbaren Daten 90
3.8.8. Binden von Parametern 93

4. Implementierung **94**
4.1. Client-Server-Architektur 94
4.2. Meilensteine 95
4.3. Spontane Ergänzungen 99
4.4. Datenbanksystem 99
4.5. Tests 101
4.6. Datenmodell 102
4.6.1. Projektressourcen 102
4.6.2. Serverseitige Persistenz 103
4.6.3. Projekt 105
4.6.4. Abfrage 106
4.6.5. Schwächen des Modells für Abfragen 109
4.6.6. Seite 111
4.7. Client 112
4.7.1. Kommunikation von Veränderungen 112

4.7.2. Existierende Implementierungen von Drag & Drop-Editoren . . . 113
4.8. Server . . . 115
4.8.1. Nutzung von Subdomains . . . 116
4.8.2. Rendervorgang . . . 116
4.9. Unerwartete Hindernisse . . . 117
4.9.1. Instabile API von Angular 2 . . . 118
4.9.2. Fehlerhafte Codegenerierung des Typescript-Compilers . . . 118
4.9.3. Verworfene Implementierung eines reinen WYSIWYG-Seiten-Editors 119

5. Fazit **125**
5.1. Erreichte Ziele . . . 125
5.2. Nicht erreichte Ziele . . . 126
5.3. Weiterentwicklung . . . 127

A. Projektbeispiele **130**
A.1. Interaktive Geschichten . . . 130
A.2. Einfacher Blog mit Kommentaren . . . 131
A.3. Pokémon Go . . . 133

B. Der Name BlattWerkzeug **137**

C. Eidesstattliche Erklärung **138**

D. Literaturverzeichnis **139**

E. CD-ROM **140**

1. Einleitung

In Deutschland scheint man Informatik als eine Hochschulangelegenheit zu betrachten. Anders ist es für den Autor dieser Thesis kaum zu erklären, dass der Informatikunterricht an Schulen zwar durchaus vorgesehen ist, der Begriff „Informatik“ dabei aber oftmals sehr weit interpretiert wird. Der Umgang mit Textverarbeitung, Tabellenkalkulation und Präsentationsmitteln ist häufig überrepräsentiert, der Horizont von Schülerinnen und Schülern[1] endet dann oft an genau dieser Stelle. Ein akkuraterer, beziehungsweise ehrlicherer Name für diese Art von Lehrveranstaltung wäre dann zum Beispiel „Office-Anwendungen“.

Dieses Problem der überrepräsentierten Inhalte wird, den Beobachtungen des Autors nach, noch dadurch gesteigert, dass im Informatikunterricht besonders verstärkt anderen Fächern zugearbeitet wird: Wenn im Deutsch-Unterricht Briefe verfasst werden sollen, haben die Schüler das eingesetzte Textverarbeitungssystem schon zu beherrschen. Eigenständige „Kerninformatik“-Inhalte mit weniger unmittelbarem Bezug zu anderen Fächern fallen dabei unter den Tisch. Das betrifft insbesondere jene Inhalte, wie sie in einer späteren Ausbildung oder einem Studium relevant wären: Themen wie Datenmodellierung, die Funktionsweise von Netzwerken oder Programmierung jenseits von Turtle-Grafiken werden hingegen, wenn überhaupt, nur sehr oberflächlich abgehandelt.

Schaut man sich darüber hinaus noch die für Informatikinhalte zur Verfügung stehenden Lehr- und Lernprogramme an (natürlich sollte Informatikunterricht auch am Computer stattfinden!), verwundert diese stiefmütterliche Behandlung noch weniger: Die Menge an verfügbarer und gepflegter Software ist ausgesprochen überschaubar. Auf der „Gegenseite“ hingegen existieren eine Vielzahl hervorragend funktionierender Büro-Anwendungen, deren Relevanz auch nicht zu leugnen ist. Und selbstverständlich ist keiner Informatiklehrkraft der Zeitaufwand zuzumuten, diese Lücke mit eigens geschriebenen Programmen zu füllen.

Diese Arbeit ist daher der Versuch, diese Lücke mit einer Lehrsoftware namens „BlattWerkzeug“[2] für zwei wichtige Teilgebiete der Informatik zu füllen: Datenmodellierung

[1] Zugunsten einer leichteren Lesbarkeit des Textes wird in dieser Thesis keine stringente Anwendung bezüglich der maskulinen und femininen Form vorgenommen. Selbstverständlich sind bei alleiniger Verwendung des generischen Maskulinums bzw. Femininums auch Personen des jeweils anderen Geschlechts mit einbezogen.

[2] Details zu dem ungewöhnlichen Namen werden in Anhang B „Der Name BlattWerkzeug“ erläutert.

und (Web-)Oberflächenentwicklung. Für diese Aufgabenbereiche ist zumindest dem Autor dieser Thesis kein verfügbares (Lern-)Programm bekannt. In einem Satz könnte man das Ziel dieser Arbeit letztendlich wie folgt zusammenfassen: „Mit BlattWerkzeug lassen sich gestützt durch *Drag & Drop-Editoren* für beliebige `SQLite`-Datenbanken *Abfragen formulieren* und *Oberflächen entwickeln*“.

Hinweis: Der im Rahmen dieser Thesis fertiggestellte Stand des Prototypen kann unter `blattwerkzeug.de` ausprobiert werden. Um nur einen kurzen Blick auf das vorläufige Arbeitsergebnis zu werfen, ist also keine lokale Installation notwendig. Da zum jetzigen Zeitpunkt allerdings noch keine Registrierung von Endanwendern implementiert worden ist, gestaltet sich der Zugriffsschutz ein wenig unbeholfen: Jedes Projekt dort verfügt über einen Testbenutzer namens `user` mit gleichlautendem Passwort. Dieser wird nur bei Speichervorgängen abgefragt, lesender Zugriff und lokale Veränderungen sind hingegen auch ohne Passwortabfrage möglich.

Aus praktischen Gründen soll an dieser Stelle die Lernsoftware „Scratch“ (siehe 2.1 „Software: Scratch“) nicht unerwähnt bleiben: Der Arbeitstitel für diese Thesis lautete „Scratch für SQL und Webanwendungen“ und weckte bei Leuten, die mit Scratch schon vertraut sind, damit durchaus gewünschte, positive Assoziationen. Wie Scratch versucht auch BlattWerkzeug viele typische und demotivierende Hürden wie Syntaxfehler, komplexe Benutzeroberflächen oder kryptische Fehlermeldungen konstruktiv auszuschließen. Kapitel 2 „Vergleichbare Arbeiten und Produkte“ beschreibt neben Scratch noch weitere, einflussreiche Inspirationsquellen.

Der wesentliche Anspruch an die im Rahmen dieser Arbeit zu erstellende Software ergibt sich also sowohl aus dem Untertitel der Arbeit, als auch der Arbeitsweise von Scratch: Es geht vorrangig um die Vermittlung von praktischen Kenntnissen zur Abfrage, Manipulation und Visualisierung von komplexen Datenbeständen in Anlehnung an die Projektideen der Lehrpläne [4] bzw. Fachanforderungen [5] für Informatik des Landes Schleswig-Holstein[3]. Auch wenn sich aktuell eine zunehmende Pluralität von Paradigmen zur Datenspeicherung abzeichnet, welche das relationale Modell ergänzen oder in Frage stellen, behandelt diese Arbeit explizit die Vermittlung von `SQL`-Kenntnissen. Abbildung 3 zeigt den Editor, mit dem in BlattWerkzeug Abfragen entwickelt werden

[3] Die exakte Verortung im Curriculum oder auch die Konzeption von konkreten Schulstunden ist hingegen nicht Teil dieser Arbeit.

Pokémon Freigelassen

Die transferierten Pokémon kommen zu Prof. Willow auf eine "Farm" und leben dort "glücklich und zufrieden bis an ihr Lebensende". Du wirst sie aber nie wieder zu Gesicht bekommen und kannst sie auch nicht besuchen.

What happens when I transfer a Pokémon to the Professor? The transferred pokemon go to a "farm" where they "live happily ever after" and you can never, ever visit or see them again.

Schau hinein!

Zurück zur Hauptseite

gefangen_id	nummer	name	spitzname
1	1	Bisasam	Bisi-Neeein
2	1	Bisasam	Dingsbums
3	1	Bisasam	Pupsi
4	4	Glumanda	Vier
5	1	Bisasam	Ösi

Freigelassenes Pokémon

Dingsbums

Löschen!

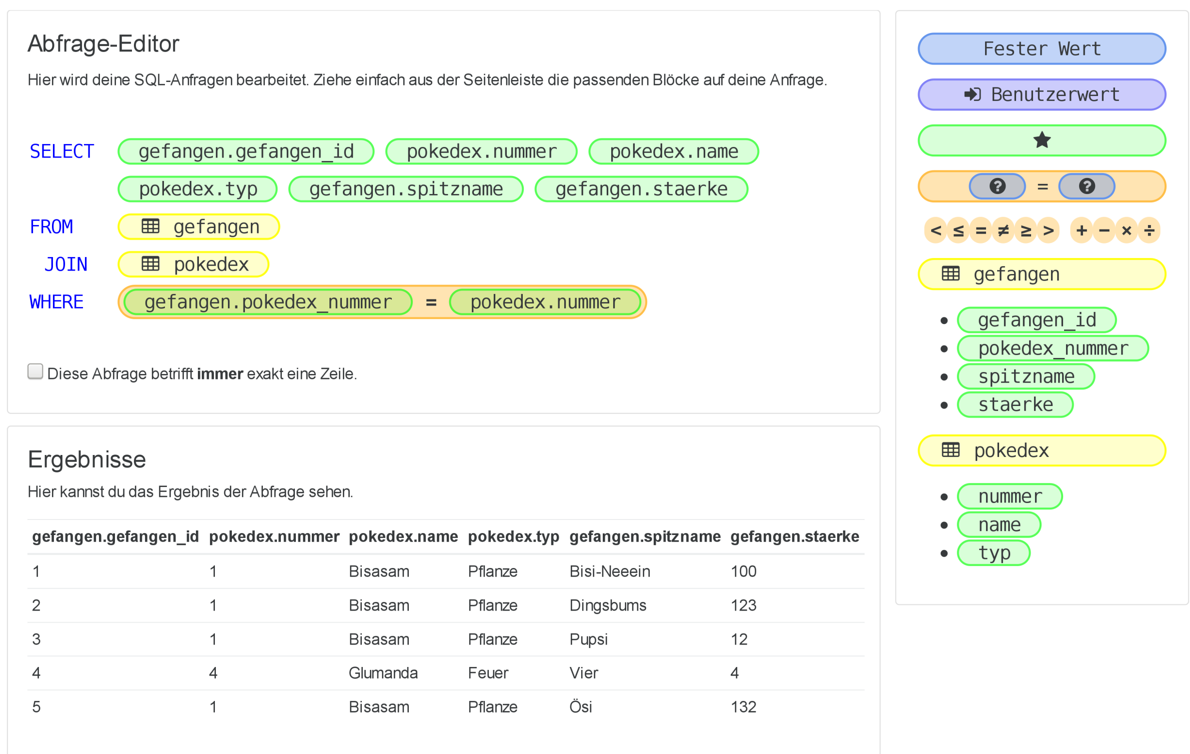
Abbildung 1: Beispiel für eine mit BlattWerkzeug erstellte Seite aus Sicht eines Endbenutzers, Abbildung 2 zeigt die Entwickleransicht. Die Abfrage aus Abbildung 3 wird für die tabellarische Auflistung verwendet.

können. Die Oberflächen werden ebenfalls in einem Drag & Drop-Editor entwickelt, ein Beispiel dafür ist in Abbildung 2 zu sehen.

Die von den Schülerinnen mit BlattWerkzeug erstellten Webauftritte werden auf den entwickelten Abfragen aufbauen und zumindest im Rahmen des in dieser Arbeit entwickelten Prototypen ein sehr einheitliches Erscheinungsbild haben: Die Ausgabe von Daten erfolgt in Tabellen, die Eingabe über Textfelder oder einfache Auswahlelemente (siehe Abbildung 1). Optische Anpassungen sind nur in sehr eng gesteckten Grenzen möglich. Und genau aus diesem Grund ist im Untertitel von einer „datenzentrierten“ Entwicklungsumgebung die Rede. In Bezug auf die Struktur der möglichen Datenmodelle ist die Flexibilität im Gegenzug nämlich ausgesprochen groß: Als Eingabe für ein BlattWerkzeug-Projekt dienen beliebige `SQLite`-Datenbank-Dateien. Der Anhang gibt dabei ein paar Hinweise, wie vielfältig die umsetzbaren Projekte daher letzten Endes sein können: Blogs, interaktive Geschichten, virtuelle Vitrinen für die eigene Sammelleidenschaft, ... Und natürlich ist diese Liste nicht abschließend zu verstehen!

Im Laufe dieser Arbeit stellt sich dabei wiederholt die Frage, wie man dieses nicht triviale Ziel einer „selbstgebauten“ Internetseite durch geschickte Abstraktionen, Vereinfachungen und Hilfestellungen einerseits erreichbar macht, andererseits aber vor lauter

Abbildung 2: Beispiel für eine mit BlattWerkzeug entwickelte Seite aus der Perspektive eines Entwicklers.



gefangen.gefangen_id	pokedex.nummer	pokedex.name	pokedex.typ	gefangen.spitzname	gefangen.staerke
1	1	Bisasam	Pflanze	Bisi-Neeein	100
2	1	Bisasam	Pflanze	Dingsbums	123
3	1	Bisasam	Pflanze	Pupsi	12
4	4	Glumanda	Feuer	Vier	4
5	1	Bisasam	Pflanze	Ösi	132

Abbildung 3: Beispiel für eine mit BlattWerkzeug formulierte `SQL`-Abfrage.

praktisch motivierenden Ergebnissen nicht die eigentlichen Lernziele der Schüler aus den Augen verliert.

Diese Arbeit gliedert sich nach der Einleitung und der Vorstellung von Inspirationsquellen in zwei wesentliche Teile: Kapitel 3 „Anforderungsanalyse“ analysiert auf Basis der vergleichbaren Arbeiten und unter Berücksichtigung eigener Ideen, wie eine solche „praktische“ Lernsoftware für datengetriebene Anwendungen aussehen könnte. Da sich die dort gesammelten Funktionen nicht alle in dem für die Thesis zur Verfügung stehenden Zeitrahmen umsetzen lassen, ist das softwaretechnische Ergebnis dieser Thesis als ein „minimum viable product“ zu verstehen. Die Umsetzung dieses Prototypen wird in Kapitel 4 „Implementierung“ beschrieben.

Kapitel 5 „Fazit“ zieht dann Bilanz und wagt einen Ausblick: Welche Ziele wurden (nicht?) erreicht, an welchen Stellen lohnt sich die Weiterentwicklung? Denn die Entwicklung von BlattWerkzeug soll nicht mit der Fertigstellung dieser Thesis eingestellt werden, sondern in einer Kooperation zwischen der Fachhochschule Wedel und der CAU Kiel weiter ausgebaut werden. Das ambitionierte Fernziel ist ein Stand, der sich für den regelmäßigen Einsatz im Unterricht eignet.

Hinweis: Lesern mit einer eher visuellen oder ergebnisorientierten Herangehensweise sei an dieser Stelle zunächst ein Blick in den Anhang A „Projektbeispiele“ empfohlen. Dort werden einige beispielhafte Projekte vorgestellt, die mit BlattWerkzeug umgesetzt worden sind.

2. Vergleichbare Arbeiten und Produkte

Andere Entwicklungsumgebungen für Datenbanken, Konzepte zur Einführung in die Programmierung und auch Generatoren für Abfragemasken gibt es in schier unüberblickbarer Zahl. Dieses Kapitel stellt einige der vefügbaren Programme oder Bücher vor und hebt dabei insbesondere den Einfluss hervor, den sie direkt oder indirekt auf BlattWerkzeug genommen haben. Sinn dieses Kapitels ist somit der „Blick über den Tellerrand", um aus der Vielzahl an verfügbaren Inspirationsquellen jene zu erwähnen, die für diese Arbeit von Bedeutung sind. Es handelt sich bewusst nicht um eine strukturierte Bewertung oder Einordnung dieser betrachteten Projekte.

2.1. Software: Scratch

Diese Masterarbeit zieht eine Menge Inspiration aus dem Scratch-Project des Massachusetts Institute of Technology (MIT)[4]. Bei Scratch handelt es sich um eine Entwicklungsumgebung speziell für Kinder und Jugendliche, deren bestimmendes Bedienkonzept eine visuelle Programmiersprache ist. Scratch lehrt den Umgang mit imperativen und ereignisorientierten Programmierkonzepten und ist damit in Bezug auf die zu vermittelnden Inhalte recht weit von dieser Thesis entfernt.

Im weiteren Verlauf der Arbeit wird dennoch deutlich werden, dass sich BlattWerkzeug an vielen Stellen sehr stark an Scratch orientiert. So nutzt auch BlattWerkzeug einen Drag & Drop-Editor um Syntaxfehler von vornherein auszuschließen und kontextsensitiv mögliche Operationen hervorzuheben. Die Verwendung eines zu diesem Zeitpunkt allerdings nicht vollständig ausgearbeiteten Farbkonzeptes nimmt sich ebenfalls Scratch zum Vorbild. Und schließlich ist auch die Zweiteilung eines Projekts in eine Besucheransicht sowie eine Entwickleransicht und der Wechsel zwischen diesen Modi eine direkte Inspiration.

Scratch baut nicht auf bestehenden Programmiersprachen auf, sondern nutzt eine Eigenentwicklung. Die Schlüsselwörter dieser Sprache, wie auch die Oberfläche des Editors, sind dabei in viele verschiedene natürliche Sprachen (Englisch, Deutsch, Spanisch, ...) übersetzt worden. Abbildung 4 zeigt, wie diese Sprache innerhalb des Scratch-Editors für einen Entwickler aussieht. Mit Scratch entwickelte Programme basieren in der Regel

[4] Verfügbar als Download- und Webversion unter `https://scratch.mit.edu/`

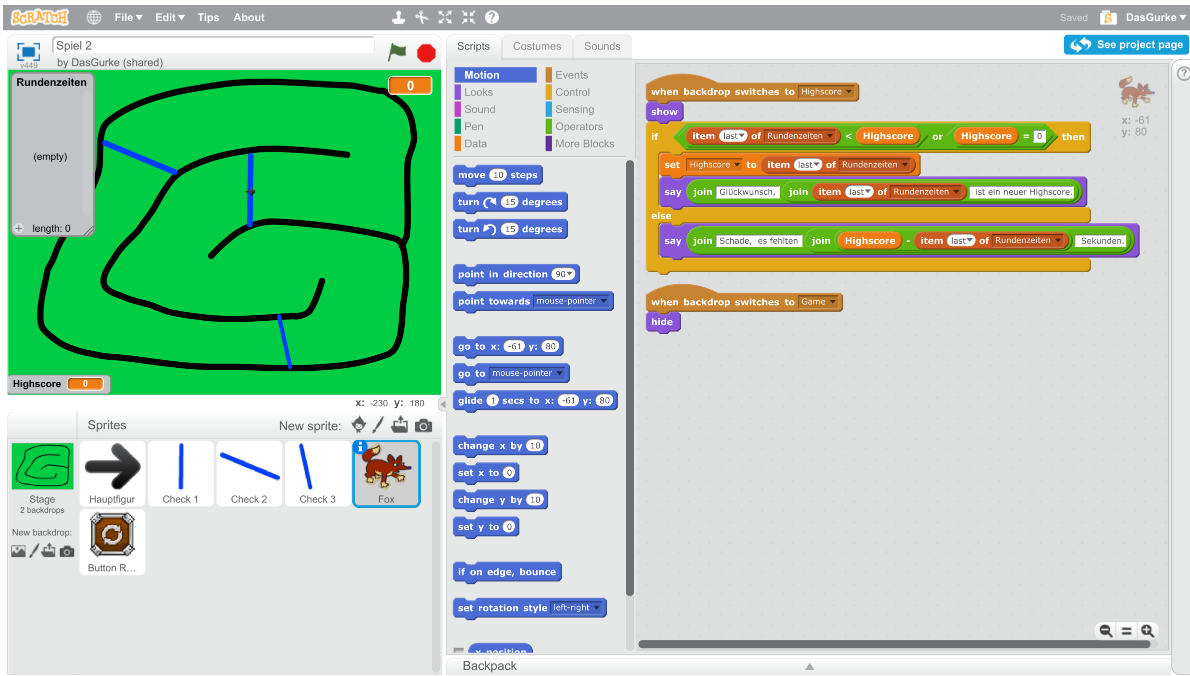


Abbildung 4: Der Scratch-Editor mit einem Beispielprojekt aus Sicht eines Entwicklers

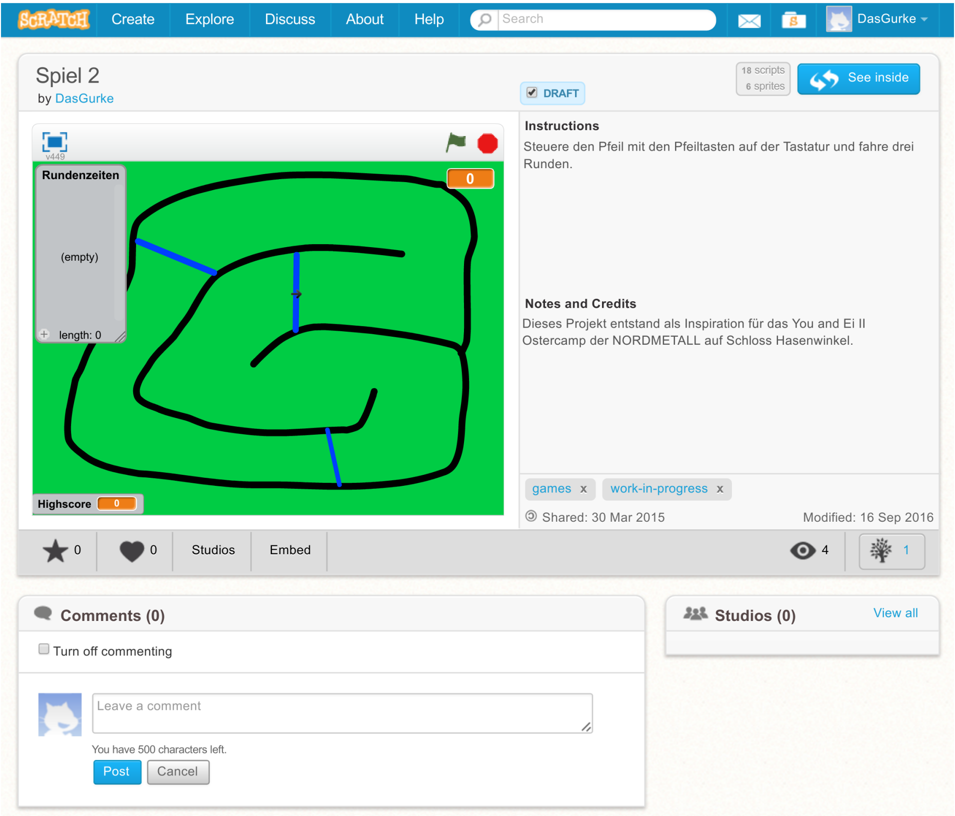


Abbildung 5: Ein Scratch-Projekt mit kurzen Handlungsanweisungen und einem Kommentarbereich in der Sicht für Endanwender

darauf, Figuren über eine Zeichenfläche zu verschieben. Dementsprechend gut eignet sich Scratch zur Entwicklung kleinerer Spiele. Die Werte von Variablen (Position, Orientierung, Größe, ... von Objekten) können sehr gut veranschaulicht und damit vergleichsweise intuitiv erfasst werden.

Abbildung 5 demonstriert, wie sich ein Projekt initial einem Endanwender präsentiert. Als Entwickler gibt man einem Projekt einen Namen, eine kurze Beschreibung und erwähnt möglicherweise Helfer und Inspirationsquellen. Besucher können über eine Kommentarfunktion Rückmeldung geben. Jenseits von den Austauschmöglichkeiten bei konkreten Projekten ist in die Webseite auch ein eigenes Forum integriert. Mit diesen sozialen Funktionen fördert Scratch aktiv den Austausch der Entwickler untereinander.

2.2. Software: App Inventor

Bei App Inventor[5] handelt es sich um eine Kollaboration des MIT mit Google und gewissermaßen um eine logische Fortführung von Scratch. Erstellt werden mit dem App Inventor nicht mehr in Webseiten einbettbare Flash-Anwendungen, sondern Apps für das Android-Betriebssystem. Wie schon Scratch läuft auch diese Anwendung in einem Webbrowser. Grundsätzlich ist auch diese Webseite in verschiedenen Sprachen verfügbar, Deutsch zählt zum gegenwärtigen Zeitpunkt allerdings nicht dazu.

Im Vergleich zu Scratch ist der App Inventor wesentlich anspruchsvoller. Die Struktur der relativ komplizierten Android-Oberflächen wird vor dem Benutzer nicht versteckt, dafür aber in einen WYSIWYG-Editor[6] verpackt (siehe Abbildung 6). Der zu schreibende Programmcode wird, wie schon bei Scratch, in einer abstrakten Sprache mit Drag &-Drop Elementen formuliert (siehe Abbildung 7). Daraus wird dann Java-Code erzeugt, welchen der Entwickler im Normalfall aber nicht zu Gesicht bekommt.

Um die Anwendungen zu testen, müssen diese auf das Handy der Entwickler geladen oder in einem Emulator ausgeführt werden. Es ist dafür erforderlich, ein (reales oder emuliertes) Gerät über die „App Inventor Companion App“ direkt mit der Webseite zu verbinden. Sobald diese Verbindung steht, werden die im Browser vorgenommenen Änderungen mit einer kurzen Verzögerung auf dem Smartphone widergespiegelt.

[5] Verfügbar unter `http://appinventor.mit.edu/`

[6] „What You See Is What You Get“ bezeichnet die visuelle Vorschau in einem Editor ohne den Umweg über einen Kompilierungsschritt.

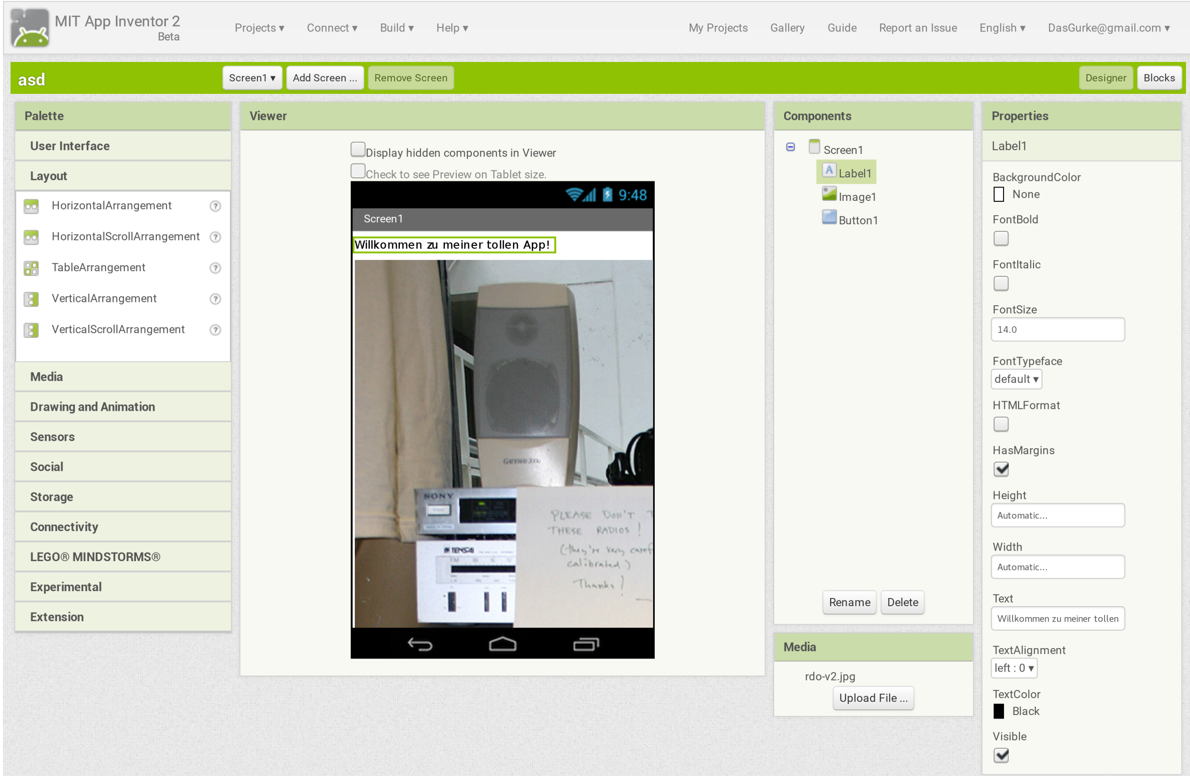


Abbildung 6: Oberflächeneditor des App Inventor

Wie Scratch fördert auch der App Inventor den Austausch von Entwicklern untereinander. Es ist möglich, die eigenen Apps in einer Gallerie auszustellen und dort Rückmeldungen von anderen zu erhalten, sowie in fremden Gallerien seinerseits Apps zu testen und zu kommentieren. Die Arbeitsergebnisse sind dabei häufig datenorientiert: Die Vorauswahl an verfügbaren Bedienelementen umfasst zwar neben typischen Bedienelementen wie Textfeldern oder Auswahllisten auch grafische Komponenten wie Bälle oder Zeichenflächen; der Umgang mit diesen Elementen ist im Vergleich zu Scratch schon durch den im direkten Vergleich deutlich langsameren Rückmeldezyklus aber wesentlich anspruchsvoller. Um korrekte Ergebnisse zu erhalten ist es notwendig, sich bereits im Voraus recht viele Gedanken zu machen.

Auch aufgrund der schieren Masse an möglichen Bedienelementen und der teilweise sehr unterschiedlichen darauf definierten Operationen ist der Komplexitätsgrad im Vergleich zu Scratch merklich gestiegen. Anders als bei Scratch kann man sich nicht mehr durch eine sehr begrenzte Anzahl von Möglichkeiten klicken und sich am Ende sicher sein, alles gesehen zu haben. Einige Operationen stehen erst im Zusammenhang mit bestimmten Bedienelementen zur Verfügung, die Suche nach einem Verb wie „kreuze an“ ist da-

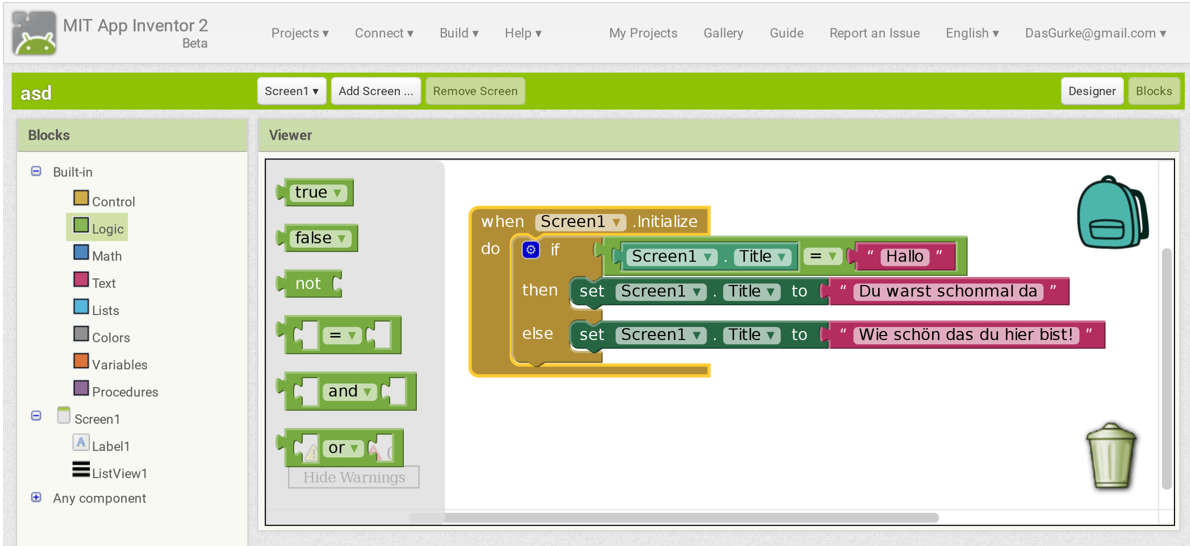


Abbildung 7: Codeeditor des App Inventor.

her nicht möglich. Man muss im Voraus wissen, welche Arten von Bedienelementen die gewünschte Funktionalität anbieten. Dieser erhöhte Komplexitätsgrad ist dabei selbstverständlich nicht nur als Nachteil zu verstehen: Man richtet sich mit dem App Inventor an eine fortgeschrittenere Zielgruppe als jene von Scratch.

2.3. Kurs: SqlZoo

Bei der Webseite SqlZoo[7] handelt es sich sowohl um einen Kurs in Form einer Sammlung von aufeinander aufbauenden Aufgaben inklusive begleitendem Lehrmaterial, als auch um eine webbasierte Software in welcher die Lernenden ihre ersten Schritte mit `SQL` machen können. Anders als in Scratch erfolgt die Entwicklung der nötigen Abfragen in einem normalen Textfeld und `SQL`. Syntaxhervorhebung oder andere unterstützende Hilfsmittel wie automatische Codeformatierung sind nicht vorhanden.

Dem Lernenden präsentiert sich das Projekt ähnlich wie ein Lehrbuch mit Übungsaufgaben. Diese sind thematisch in Kapitel untergliedert und nutzen in der Regel einen sehr überschaubaren Datenbestand. Die Ergebnisse werden neben dem Textfeld angezeigt und können mit dem korrekten Ergebnis verglichen werden (Abbildung 8). So können Lernende ihre Ergebnisse selbständig kontrollieren.

[7] Verfügbar unter `http://sqlzoo.net/`

Nach den Lerneinheiten folgt dann eine Serie von Quizfragen (Abbildung 9). Dabei handelt es sich um Multiple-Choice-Aufgaben, es müssen also keine `SQL`-Abfragen geschrieben werden. Primär wird bei diesen Fragen eine Zuordnung verlangt: Dem Lernenden wird wahlweise eine Ergebnismenge oder eine Abfrage gezeigt und es muss die jeweils passende Abfrage oder Ergebnismenge dazu identifiziert werden. Weitere Aufgabentypen beschäftigen sich zum Beispiel mit der Wahl der korrekten `SQL`-Formalisierung zu einer natürlichsprachlich formulierten Anfrage.

Eine Besonderheit dieses Angebots ist die sehr breite Unterstützung verschiedener Datenbanksysteme. Als Lernender kann man die Aufgaben wahlweise mit dem Microsoft SQL Server, Oracle, MySQL, DB2, Ingres oder PostgreSQL lösen. SqlZoo wird ebenfalls in mehreren Sprachen angeboten, die deutsche Übersetzung ist allerdings nicht durchgehend verfügbar.

Es ist mit dem SqlZoo prinzipiell auch möglich, mutierende Abfragen auszuführen. Diese werden im Normalfall allerdings nicht über eine einmalige Anzeige hinaus persistiert. Angemeldete Benutzer können zwar auch eigene Tabellen anlegen, agieren damit aber immer in einem globalen Namensraum ohne differenzierte Zugriffskontrollen: Es ist jedem bearbeitenden Nutzer möglich beliebige Datenbestände zu verändern. Alle mutierenden Operationen - es wird zum Beispiel auch `CREATE TABLE` behandelt - sind dabei aktuell lediglich als Referenz verfügbar und nicht Teil des Kursangebotes mit dazu ausgearbeiteten Quizaufgaben.

Auch wenn es durch den Wiki-Unterbau von SqlZoo den Lernenden theoretisch möglich wäre eigene Projektideen innerhalb der Webseite umzusetzen, ist dies eindeutig nicht das Ziel des Angebotes. Anders als bei Scratch oder dem App Inventor steht am Ende dieses Angebotes also keine eigene Software, sondern Erkenntnisgewinn.

2.4. Software: Jekyll

Bei Jekyll[8] handelt es sich um ein quelloffenes Framework zur Erstellung statischer Webseiten. Der Begriff „statisch“ bezeichnet in diesem Kontext das Ergebnis der Kompilierung: `HTML` Seiten, die von jedem Webserver unverändert ausgegeben werden können. Technisch gesehen setzt Jekyll auf die `HTML`-Templatingsprache Liquid auf und stellt während der Generierung einer Seite eine Vielzahl von hilfreichen Metadaten bereit. Diese

[8] Dokumentation verfügbar unter `http://jekyllrb.com/`

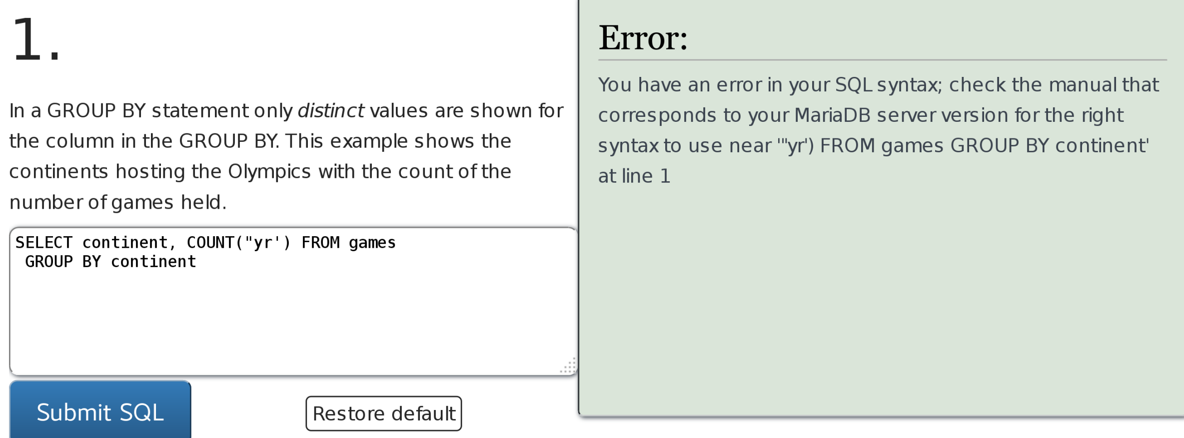


Abbildung 8: Technische Fehlermeldung einer Datenbank aufgrund eines Syntaxfehlers in der Eingabemaske von SqlZoo.

3. Pick the code that shows the amount of years where no Medicine awards were given

```
SELECT COUNT(DISTINCT yr) FROM nobel
 WHERE yr IN (SELECT DISTINCT yr FROM nobel WHERE subject <> 'Medicine')
```

```
SELECT COUNT(DISTINCT yr) FROM nobel
 WHERE yr NOT IN (SELECT DISTINCT yr FROM nobel WHERE subject = 'Medicine')
```

```
SELECT DISTINCT yr FROM nobel
 WHERE yr NOT IN (SELECT DISTINCT yr FROM nobel WHERE subject LIKE 'Medicine')
```

```
SELECT COUNT(DISTINCT yr) FROM nobel
 WHERE yr NOT IN (SELECT DISTINCT yr FROM nobel WHERE subject NOT LIKE 'Medicine')
```

```
SELECT COUNT(yr) FROM nobel
 WHERE yr NOT IN (SELECT DISTINCT yr FROM nobel WHERE subject = 'Medicine')
```

Abbildung 9: Quizfrage: Zu einer natürlichsprachlichen Aussage muss die korrekte SQL-Formalisierung gewählt werden.

Metadaten werden entweder implizit aus Dateinamen gewonnen oder in `YAML`-Notation im Kopf einer Datei definiert.

In der Standardkonfiguration eignet sich Jekyll vor allem zum Betreiben einer Blog-artigen Seite ohne großen Installationsaufwand. Alle Dateien die sich in einem speziellen Ordner `_posts` befinden, werden als Datenquelle für Blogeinträge behandelt. Ein solcher Eintrag könnte dabei aussehen wie in Listing 1, typischerweise würden natürlich mehrere solcher Dateien für Beiträge existieren. Bei dem Bereich zwischen den `---` handelt es sich um das so genannte „`YAML`-Frontmatter“. Diese Metadaten stehen dem Entwickler im Kontext dieser Seite zur Verfügung. Um diese in die Seite einzubetten, werden sie als Liquid-Ausdruck in geschweifte Klammern geschrieben.

Einige der Metadaten werden von Jekyll ausgewertet, zum Beispiel `layout`. Dabei handelt es sich um das Liquid-Template, welches einen solchen Beitrag darstellen soll. Wie ein solches Template aussieht, zeigt Listing 2. Auch dieses Layout wird seinerseits wieder in ein anderes Layout eingebettet, der Ausdruck `{{ content }}` bezeichnet dabei den Ort an dem diese Einbettung stattfindet. Listing 3 zeigt, wie die Kette von Einbettungen in einer letzten Vorlage endet. Ziel dieser Einbettung ist die Vermeidung von Redundanzen: Um zum Beispiel auf allen Seiten eine neue `CSS`-Datei zu referenzieren, muss in diesem Beispiel nur Listing 3 angepasst werden.

Die Metadaten werden von Jekyll in unterschiedliche Namensräume gruppiert. Globale Daten werden in einer speziellen `YAML`-Datei hinterlegt und können über das Präfix `site` adressiert werden. Daten zu spezifischen Seiten erhalten das Präfix `page`.

Um auf einer Hauptseite alle Beiträge anzuzeigen, stellt Jekyll im `site`-Namensraum eine spezielle Liste names `posts` zur Verfügung. Diese können dazu genutzt werden, alle Blogbeiträge auf einer Hauptseite anzuzeigen. Listing 4 zeigt, wie der Code für eine solche Seite aussehen könnte. Die Eigenschaften `page.url` und `page.excerpt` werden dabei automatisch von Jekyll generiert.

Die Generierung von Webseiten in BlattWerkzeug ist stark von Jekyll inspiriert. Die offensichtlichste Gemeinsamkeit ist die Nutzung der gleichen Templating-Engine „Liquid“, aber auch das Datenmodell und dessen Aufteilung in verschiedene Namensräume wurde von Jekyll inspiriert. In einer der ersten Prototypen handelte es sich bei BlattWerkzeug sogar um ein Plugin für Liquid, welches im Kopfbereich definierte SQL-Abfragen auswertete und im Kontext der Seiten zur Verfügung stellte. Die statische Natur der von Jekyll

```
---
layout: post
title: So funktioniert Jekyll
categories: [simple,beginner]
author: Marcus
test: 3.141
---
Dies ist ein einfacher Blogeintrag mit wenigen Metadaten. Das Datum des
Beitrages wird aus dem Dateinamen generiert. {{ page.test }} ist ein
Wert, der testweise ausgegeben wird.
```

Listing 1: Ein Blogbeitrag mit Metadaten für Jekyll

```
---
layout: default
---
<h2>{{ page.title}} <small>von {{ page.author}}</small></h2>
<ul>
  {% for category in page.categories %}
    <li>{{ category }}</li>
  {% endfor %}
</ul>
<div class="content">
  {{ content }}
</div>
```

Listing 2: Beispieltemplate für Blogbeiträge für Jekyll

```
<html>
  <head><title>{{ site.title }} - {{ page.title }}</title></head>
  <body>
    {{ content }}
  </body>
</html>
```

Listing 3: Template mit `HTML`-Rumpf für Jekyll

```
---
layout: default
---
<h1>{{ site.title }}</h1>
{% for post in site.posts %}
  <div>
    <a href="{{ post.url }}">{{ post.title }}</a>
    {{ post.excerpt }}
  </div>
{% endfor %}
```

Listing 4: Hauptseite mit Auszügen aller Beiträge für Jekyll

generierten Seiten disqualifiziert es allerdings als Kandidat für eine softwaretechnische Basis von BlattWerkzeug.

2.5. Software: Visual Studio Lightswitch

Diese Variante des Microsoft Visual Studio ist auf die Entwicklung von datenorientierten Geschäftsanwendungen ausgelegt. Sie erlaubt eine schnelle Integration von heterogenen Datenquellen und erstellt Anwendungen, die entweder als Silverlight-Desktopanwendung oder im Browser laufen.

Visual Studio Lightswitch richtet sich vornehmlich an Informatik-affine Mitarbeiter, die jedoch im Regelfall nicht zwingend über eine formale Informatik-Ausbildung verfügen. Es kann gewissermaßen als der logische nächste Schritt nach dem Datenbanksystem Microsoft Access gesehen werden. Die Programmierung dieser Anwendungen erfolgt in den normalen Editoren für JavaScript oder C# des Visual Studio. Beide Sprachen sind prinzipiell im vollen Umfang verfügbar, es existiert keine gesonderte Unterstützung, um Anfängern die ersten Schritte zu erleichtern oder typische Fehler zu lokalisieren. Eine Vereinfachung ergibt sich durch die spezialisierte Programmierschnittstelle, welche die Verknüpfung unterschiedlicher Datenquellen mit der Oberfläche erleichtern.

Es gibt unterschiedliche Arten und Weisen, um innerhalb einer Lightswitch-Anwendung auf Daten zuzugreifen. Im Rahmen dieser Arbeit ist aber lediglich der Umgang mit Datenbanken von Bedeutung. Für diese existiert ein eigener Editor (Abbildung 10), welcher es erlaubt die Spalten einzelner Tabellen zu editieren und Beziehungen einzufügen. Die benötigten C#-Klassen werden dann automatisch erzeugt und mittels eines Objekt-Relationalen-Mappers (`ORM`) auf die Struktur der Datenbank gemappt. Diese Vorgehensweise führt dabei zu einer Einschränkung, die im Lehrkontext zumindest ärgerlich ist: `m:n`-Beziehungen zwischen Objekten können nicht erstellt werden, sondern müssen manuell durch zwei `1:n`-Beziehungen mit einer „künstlichen“ Entität in der Mitte abgebildet werden.

Die Oberflächen werden in einem Editor erstellt, welcher die Hierarchie der Bedienelemente in einem Baum visualisiert (Abbildung 11). Es stehen abhängig von der Zielplattform (Browser oder Desktop) im Detail unterschiedliche, aber in der Programmierung sehr ähnliche Bedienelemente bereit. Wichtigstes Element ist dabei die sehr mächtige Visualisierung von tabellarischen Daten.

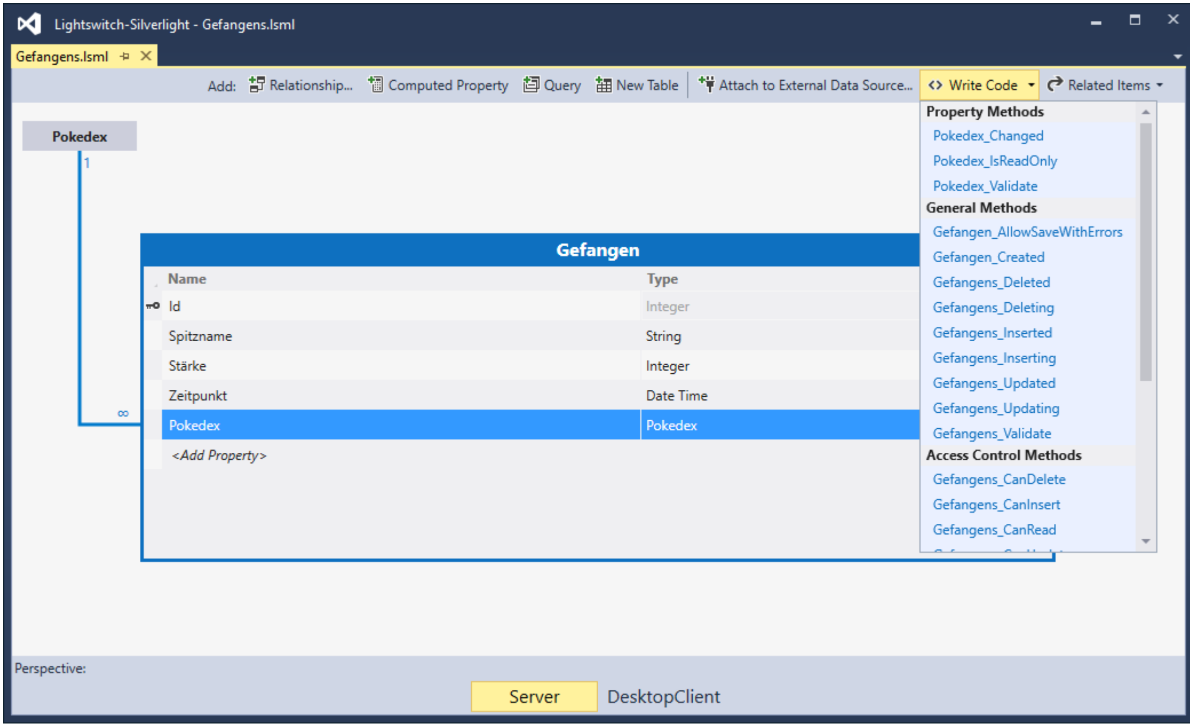


Abbildung 10: Schemaeditor von Visual Studio Lightswitch

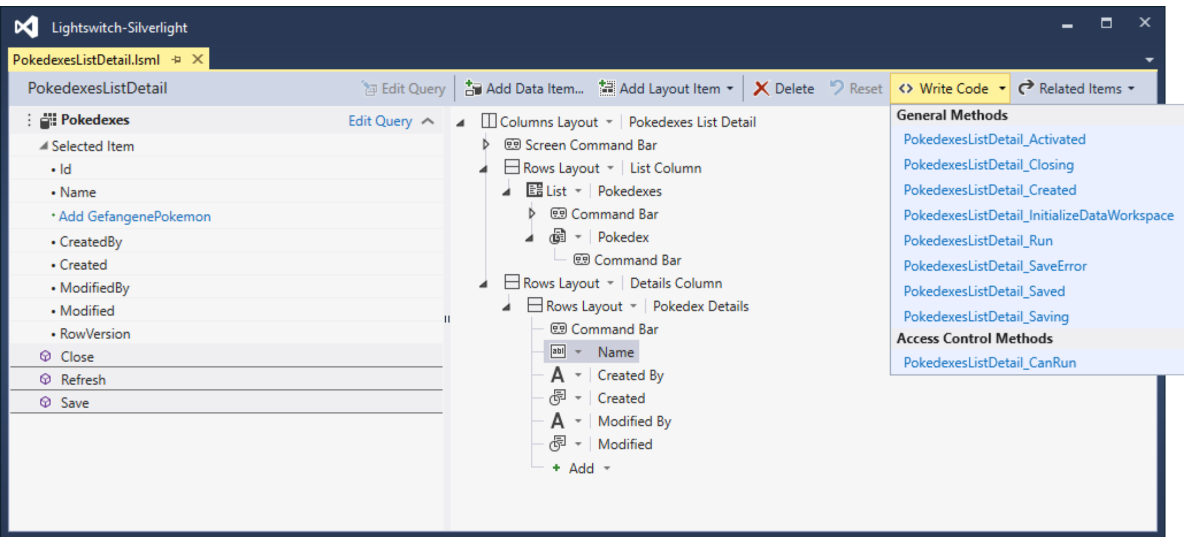


Abbildung 11: Oberflächeneditor von Visual Studio Lightswitch

Beide Editoren erlauben die Implementierung von speziellen Methoden, welche den `CRUD`-Zyklus[9] oder spezielle Ereignisse innerhalb der Oberfläche abbilden. Es ist zwar auch möglich, Code außerhalb dieser vordefinierten Struktur zu schreiben, dieser kann dann aber nicht unmittelbar in den Editoren referenziert werden.

Ein Lightswitch-Projekt gliedert sich in zwei wesentliche Unterstrukturen: Datenquellen und Beschreibungen von Oberflächen. Diese werden auf zwei Projekte für Server und Client aufgeteilt, das Datenmodell ist jedoch transparent auf beiden Seiten verfügbar. Diese klare Trennung zwischen Datenmodell und Visualiserung findet sich auch bei Blatt-Werkzeug wieder.

2.6. Buch: Die Macht der Abstraktion

Bei diesem Buch von Herbert Klaeren und Michael Spaerber wird ein sehr von Datenstrukturen geprägter Ansatz zur Einführung in die Programmierung gewählt. Die Autoren beschreiben die gängigen Programmiersprachen aufgrund ihrer umfangreichen Syntax als ungeeignet für die Einführung in die Programmierung [3, S. 2].

Folglich unterrichtet das Buch auch nicht Java, C#, Python, Ruby oder Pascal. Stattdessen kommt der Scheme-Dialekt „Racket" zum Einsatz. Bei Scheme handelt es sich wiederum um eine der wesentlichen Varianten von Lisp, einer Programmiersprache, deren Syntax fast exklusiv auf der Verwendung von runden Klammern zur Beschreibung von Tupeln aufbaut. Durch diese sehr reduzierte Syntax versprechen sich die Autoren die Vermeidung von typischen syntaktischen Stolperfallen, mit denen Lernende zu Beginn oft Schwierigkeiten haben.

Die Anpassung an eine für Lernende gut geeignete Umgebung geht allerdings über diese vergleichsweise simple Syntax hinaus: Der Sprachumfang von Racket selbst wird im Laufe des Buches mehrfach erweitert. Abbildung 12 zeigt den Auswahldialog der „Dr Racket"-Umgebung, in welchem auch die zum Buch passenden Sprachumfänge verfügbar sind.

[9] Operationen auf Datensätzen lassen sich, unabhängig von der Art der Implementierung, in die vier Phasen **C**reate, **R**ead, **U**pdate & **D**elete einteilen.

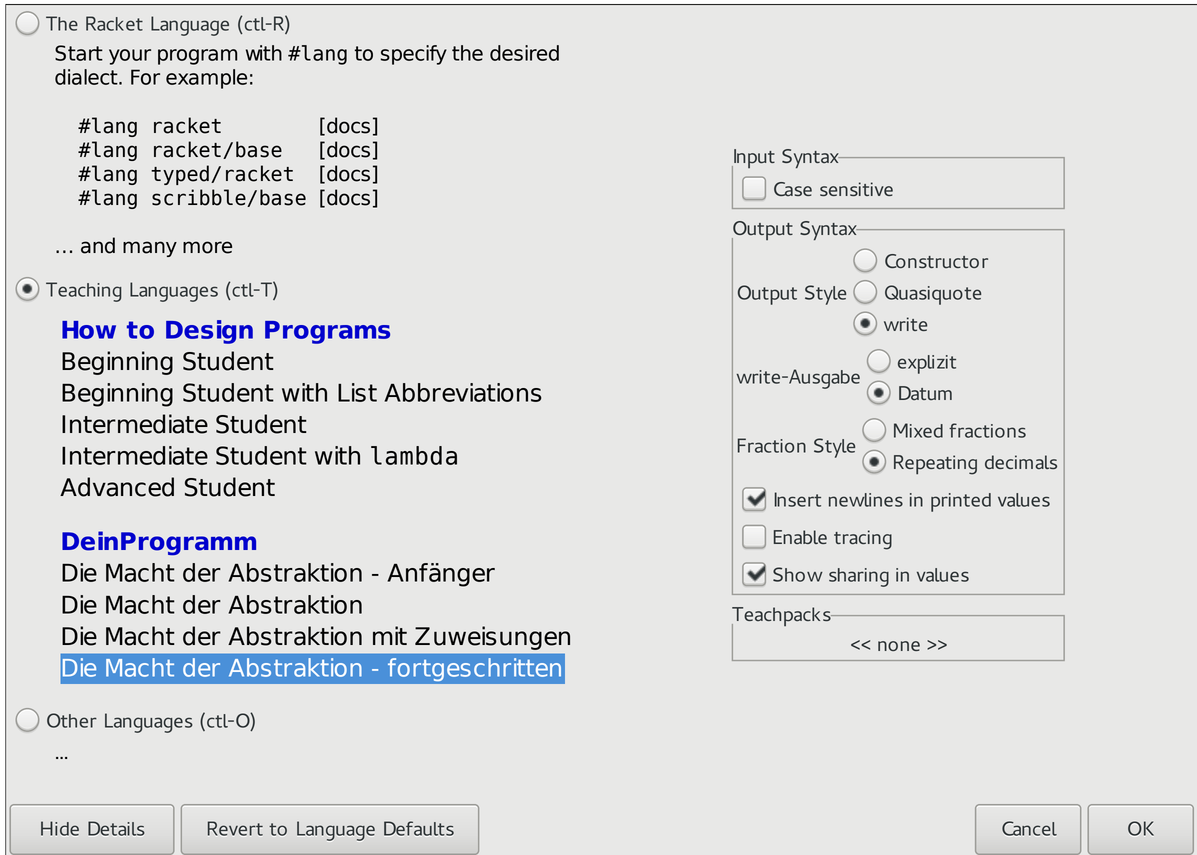


Abbildung 12: Verfügbare Sprachniveaus in „Dr Racket“

2.7. Software: MySQL Workbench

Die MySQL Workbench ist das hauseigene Management- und Verwaltungsprogramm des gleichnamigen Datenbanksystems. Es unterstützt Programmierer mit speziellen Funktionen zur Modellierung von Datenbanken oder beim Entwickeln von Abfragen zu existierenden Datenbeständen. Systemadministratoren haben mit der Workbench Zugriff auf Performance-Indikatoren wie Caching-Fehlerraten, Datendurchsatz oder sonstige Auslastungsmerkmale.

Anders als das Visual Studio Lightswitch unternimmt die MySQL Workbench keinerlei Versuche technische Details vor dem Anwender zu verstecken. Es handelt sich im wesentlichen um eine grafische Benutzerschnittstelle für jene Daten, die ebenso von der Programmierschnittstelle bereitgestellt werden.

Da es sich um ein spezialisiertes Programm eines Datenbanksystems handelt, bietet der Schema-Designer (Abbildung 13) keinerlei Unterstützung für konzeptuelle Model-

lierung (z.B. mit Entity-Relationship-Modellen). Er beschäftigt sich ausschließlich mit der Darstellung oder Modifikation von physikalischen Modellen. Im Editor stehen zwar Komfortfunktionen für die Modellierung von Beziehungen aller typischen Kardinalitäten zur Verfügung, diese werden aber unmittelbar auf Fremdschlüssel und gegebenenfalls Kreuztabellen abgebildet. Von `m:n`-Beziehungen wird folglich nicht abstrahiert, sondern stattdessen die gängige Abbildung in Form einer Kreuztabelle automatisch vorgenommen.

Der Editor für Abfragen bietet alle typischen Funktionen einer Entwicklungsumgebung (Abbildung 14): Dem Entwickler werden kontextsensitiv Vervollständigungsvorschläge angeboten, die Struktur der involvierten Tabellen ist in einer Seitenleiste sichtbar. Das Ergebnis der Abfrage wird in einem variabel großen Bereich unter dem Texteditor angezeigt. Und unter diesem Bereich wiederum ist eine Historie der zuletzt ausgeführten Abfragen verfügbar.

Die Workbench versucht sowohl syntaktische (zum Beispiel fehlende Klammern) als auch logische (zum Beispiel nicht vorhandene Spalten) Fehler optisch hervorzuheben. Dafür werden die Abfragen anscheinend aber nicht gegen den Server validiert. Zumindest entstehen gelegentlich Inkonsistenzen in beide Richtungen: Es werden möglicherweise Fehler im Editor angezeigt, welche auf dem Server nicht auftreten. Auch der umgekehrte Fall kann eintreten: Tatsächlich aufgetretene Fehler auf dem Server werden nicht mit einer entsprechenden Markierung im Editor versehen.

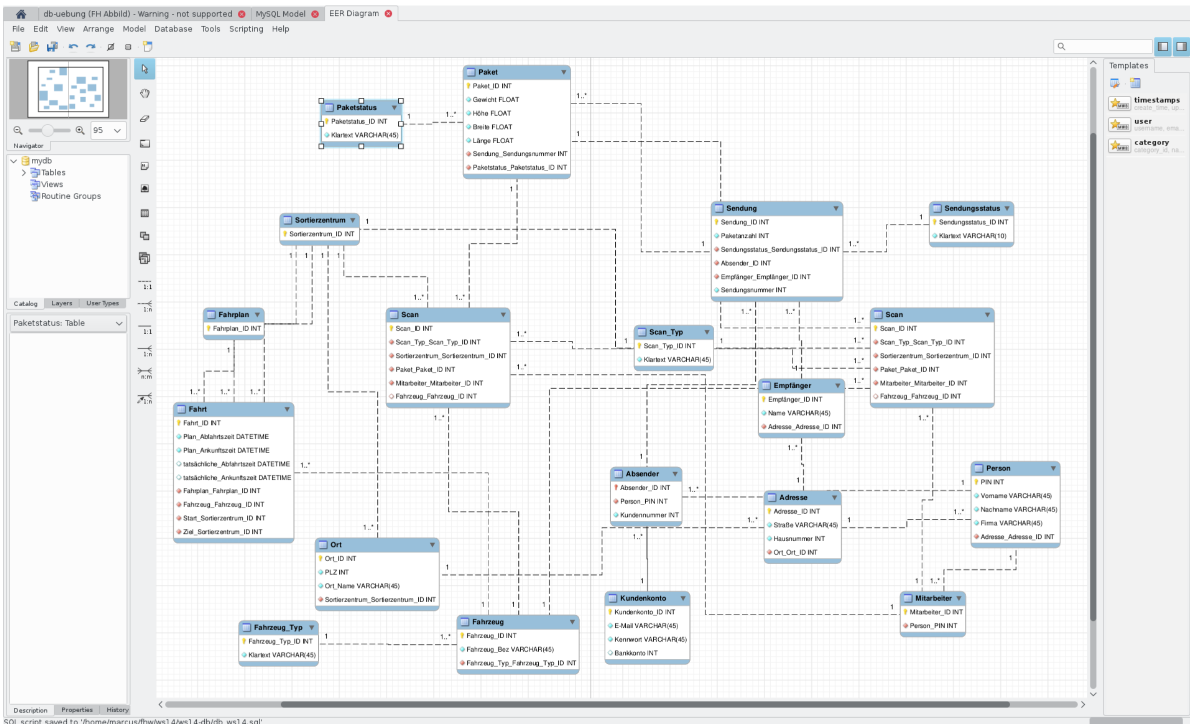


Abbildung 13: Visueller Schema-Editor der MySQL Workbench

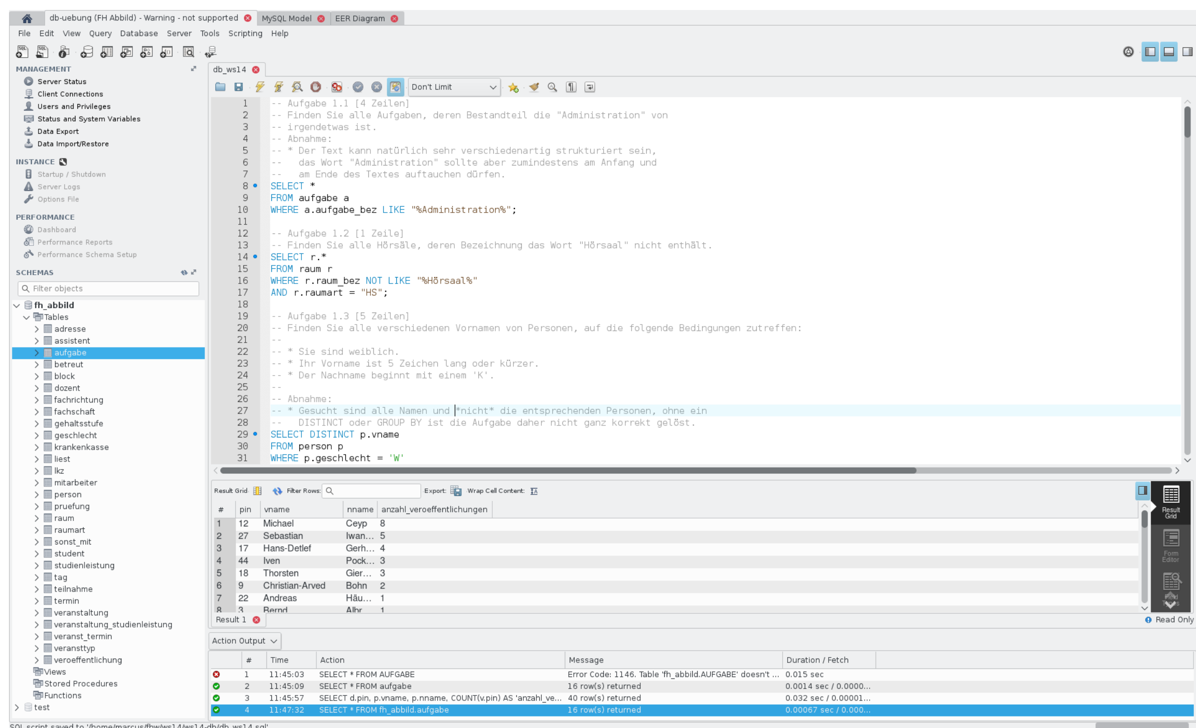


Abbildung 14: Abfrageeditor der MySQL Workbench

3. Anforderungsanalyse

Dieses Kapitel beschreibt die generellen Anforderungen an die Entwicklungsumgebung. Dabei werden auch einige theoretische Hintergründe betrachtet, speziell mögliche Reduktionen des `SQL`-Sprachumfangs für die Vermittlung von Datenbankkenntnissen. Ebenfalls thematisiert werden unterschiedliche Vorgehensweisen, um Schüler bei der Entwicklung von Oberflächen sinnvoll anzuleiten. Den Schwerpunkt bildet die Beschreibung der beiden wichtigsten Komponenten der Entwicklungsumgebung: die spezialisierten Editoren für `SQL` und Oberflächen.

Hinweis: Der Verzicht auf die technischen Hintergründe der Implementierung soll dieses Kapitel auch für Informatik-Interessierte ohne softwaretechnischen Hintergrund verständlich halten, anstatt die konzeptionellen Überlegungen mit den Details der Realisierung zu überlagern. Die Planung konkreter Umsetzungsstrategien und die Diskussion von softwaretechnischen Details, werden in Kapitel 4 „Implementierung“ besprochen.

BlattWerkzeug geht im Vergleich zu Scratch und dem App Inventor in Bezug auf die gelehrten Programmiersprachen einen anderen Weg: Es lehrt unmittelbar den Umgang mit `SQL` und `HTML` ohne dabei die Syntax zu verstecken, allerdings in gegebenenfalls reduzierten Umfängen (siehe 3.5 „Konzept für sinnvolle Teilmengen von `SQL`“ und 3.7 „Konzept für Oberflächen“). Die Entwicklung eigener Programmier- oder Auszeichnungssprachen für die Lehre mag zwar die Einbindung spezieller didaktischer vorteilhafter Konzepte ermöglichen, geht im Rahmen dieser Arbeit aber auch mit zwei gravierenden Nachteilen einher: Zum Einen ist die Entwicklung einer konsistenten Programmiersprache alles andere als trivial (von zweien ganz zu schweigen); Zum Anderen erschwert eine solche Insellösung den Übergang zu „richtiger“ Programmierung in der „echten“ Welt außerhalb dieser Lehrumgebung. Abgesehen davon handelt es sich bei `SQL` und `HTML` um vergleichsweise einfache Sprachen, die sich darüber hinaus gut im Umfang reduzieren lassen. Aus diesen Gründen setzt BlattWerkzeug auf bestehende und etablierte Standards auf, anstatt eine Eigenentwicklung in Betracht zu ziehen.

Das in diesem Kapitel angestrebte Konzept wird aufgrund der Endlichkeit der im Rahmen einer Master-Thesis zur Verfügung stehenden Zeitspanne nicht vollständig umgesetzt werden können. Trotzdem ist eine möglichst vollständige Erfassung von Anfor-

derungen unerlässlich, um bei Fortführung der Entwicklung nicht aufgrund von unberücksichtigten Anforderungen große Umbauarbeiten vornehmen zu müssen. Die Kapitel 3.6 „Drag & Drop-Editor für `SQL`“ & 3.8 „Drag & Drop-Editor für Oberflächen“ stellen streng genommen schon einen kleinen Vorgriff dar: Statt die Konzepte für die Gestaltung der Oberflächen nur zu beschreiben, zeigen sie vereinzelt Screenshots, die während oder am Ende der Entwicklung des Prototypen entstanden sind.

Da der Zweck dieser Arbeit die Konzeption und Umsetzung einer Software ist, deren Zweck wiederum die Entwicklung anderer Software ist, bedarf es zunächst eines einheitlichen Verständnisses für die Bezeichnungen der beteiligten Personen und Entitäten:

(Schüler-)Entwicklungsumgebung, BlattWerkzeug

Bezeichnet die von den Lernenden zu nutzende Software, die im Rahmen dieser Arbeit erstellt wird. Sofern Verwechslungen mit anderen Entwicklungsumgebungen (engl. “integrated development environment”, IDE) auftreten könnten, wird explizit das Präfix “Schüler” genutzt. Diese Unterscheidung wird insbesondere bei Vergleichen mit gängiger Entwicklungssoftware relevant sein.

(Schüler-)Projekt

Bezeichnet die von den Lernenden unter Nutzung der Schülerentwicklungsumgebung erstellte Software. Teil eines solchen Projektes sind unter anderem das Datenbankschema, die verschiedenen Abfragen und die gestaltete Benutzeroberfläche.

Entwickler

Bezeichnet Personen, die mit der “entwickelnden” Benutzeroberfläche eines Schülerprojekts interagieren.

Endanwender

Bezeichnet Personen, die mit der von den Entwicklern erstellten “normalen” Benutzeroberfläche eines Schülerprojekts interagieren, nicht jedoch mit der Schülerentwicklungsumgebung selbst.

3.1. Zielgruppe

Mit BlattWerkzeug sollen Schülerinnen ab der Mittelstufe angesprochen werden. Die im folgenden genannten Aspekte bilden ein sehr oberflächliches Profil hilfreicher Vorkenntnisse, an denen man sich als Lehrkraft für die Fragestellung „Sind meine Schüler

bereit für BlattWerkzeug?" orientieren könnte. Dieses Profil ist dabei allerdings nicht als striktes Anforderungsprofil zu verstehen: Grundsätzlich lassen sich diese Kenntnisse auch am Beispiel von BlattWerkzeug vermitteln. Erfahrungen mit Datenbanken oder Webentwicklung sind hingegen nicht Bestandteil dieses Profils, schließlich versteht sich BlattWerkzeug als eine Lernumgebung für genau diese Technologien.

Grundlegende PC-Anwenderkenntnisse
: Die Schüler müssen in der Lage sein, einen Browser zu bedienen und mit Begriffen wie „Speichern" und „Klicken und Ziehen" vertraut sein. Kenntnisse über Speicherorte im Dateisystem sind jedoch nicht notwendig, BlattWerkzeug abstrahiert die konkreten Speicherorte.

Tabellenkalkulation
: Kenntnisse über den Umgang mit Tabellenkalkulationsprogrammen sind keine zwingende Voraussetzung, aber eine sinnvolle Vorstufe, um die Strukturierungsmöglichkeiten von Datenbeständen zu verstehen. Sofern sich die Schüler innerhalb eines solchen Programms auch schon mit Funktionen wie `SUMME` beschäftigt haben, kann auf diesem Wissen aufgebaut werden.

Bedienelemente
: Wenn die Schüler mit BlattWerkzeug eigene Oberflächen entwickeln sollen, brauchen sie grundlegende Vorstellungen über die Funktionsweise und Parameter einiger Bedienelemente: Mit Links werden Inhalte verknüpft, Knöpfe lösen Aktionen aus, Textfelder stellen Werte zur Verfügung, eine Combobox erlaubt die Auswahl von einem Wert aus vielen Werten ... Ohne die mit einem Bedienelement durchgeführte Aktion zu verstehen, erschließt sich deren Verwendung in `HTML` nur sehr umständlich.

Englische Vokabeln
: In der initialen Version wird BlattWerkzeug die Syntax von `SQL` oder `HTML` zwar vereinfachen und augmentieren, aber nicht übersetzen. Den Schülern sollte die Bedeutung von Fragmenten wie `WHERE` oder `<input>` also geläufig sein.

3.2. Grundprinzipien

Nach der Betrachtung der zu bedienenden Zielgruppe und der Beschäftigung mit bereits existierenden Alternativen ist es zunächst sinnvoll ein paar allgemeine Grundprinzipien

zu formulieren. Diese Prinzipien bilden die Philosophie hinter der Schülerentwicklungsumgebung ab und dienen als Leitfaden für Designentscheidungen. Praktisch erlaubt das vor allem eine relativ akkurate Abschätzung, ob sich die Implementierung einer bestimmten Idee lohnt und wie sie gegebenenfalls zu priorisieren ist. Die Sortierung dieser Prinzipien ist entsprechend ihrer Bedeutung absteigend sortiert, das wichtigste Prinzip wird also zuerst genannt.

Semantik vor Syntax

Den Lernenden sollen kontextsensitiv sinnvolle Operationen angeboten werden, optimalerweise mit einer kurzen Erläuterung, warum gerade nur diese Teilmenge an Operationen möglich ist. Die eigentliche Programmierung erfolgt dann durch die Kombination von Bausteinen, ähnlich wie bei der Lernsoftware „Scratch“. Durch kontinuierliches Feedback der Entwicklungsumgebung sollen die Lernenden in die Lage versetzt werden, auch ohne ständige Rückversicherung bei der Lehrkraft eigene Ansätze zu erproben. Syntaxfehler, ein typisches Problem für viele Anfänger, werden durch dieses Vorgehen konstruktiv verhindert.

Motivation durch praktisch vorzeigbare Ergebnisse

Der Einstieg in die Programmierung ist oftmals von relativ langweiligen Programmen geprägt, häufig textbasierten Konsolenanwendungen, welche sich nicht gut im Freundes- oder Bekanntenkreis präsentieren lassen. Im Sonderfall der Vermittlung von `SQL`-Kenntnissen ist das Ergebnis der investierten Arbeit sogar überhaupt nicht sinnvoll zu demonstrieren. Die erstellten Abfragen stehen isoliert für sich und sind häufig auch nur in der Entwicklungsumgebung der jeweiligen Datenbank ausführbar. Mit der im Rahmen dieser Arbeit zu erstellenden Software sollen sich hingegen praktisch relevante, allerdings sehr datenorientierte Programme umsetzen lassen. Diese verfügen über von den Lernenden zusammengestellte Eingabemasken, um Daten einzufügen oder zu manipulieren sowie verschiedene Ausgabesichten, um den Datenbestand sinnvoll zu präsentieren.

Schrittweise komplexere Benutzeroberfläche der Entwicklungsumgebung

Konventionelle Entwicklungsumgebungen sind Programme von Profis für Profis und bieten einen dementsprechend ausgerichteten Funktionsumfang. Gerade wenn man aber dabei ist etwas Neues zu lernen, kann es sinnvoll sein, die Menge der möglichen Optionen zu beschränken. In diesem Sinne sollte die Lehrkraft die Möglichkeit haben, den Funktionsumfang der Entwicklungsumgebung für Schüler gezielt zu reduzieren. Es sollten sich also Funktionen der Entwicklungsumgebung ausblenden

lassen, wenn dies aus didaktischen Gründen sinnvoll erscheint. Zum Beispiel könnten bestimmte Bedienelemente oder Bestandteile von `SQL` ausgeblendet werden, wenn diese noch nicht behandelt worden sind. Dieser Ansatz die Oberfläche insgesamt anpassbar zu gestalten, wird auch von anderen Lernprogrammen verfolgt, zum Beispiel der Mathematik-Lernsoftware GeoGebra[10].

Einfache Inbetriebnahme

Eine initiale Hürde jeder (Lern-)Software ist deren Installation, insbesondere bei Programmen aus dem Datenbankumfeld. Die Inbetriebnahme der für Server konzipierten Programme auf privaten, „normalen" Rechnern führt immer wieder zu Problemen aufgrund von unerfüllten Abhängigkeiten, fehlenden Rechten beim Starten von Systemdiensten oder bei Dateizugriffen. Die zunehmende Heterogenität von Geräteklassen, also die Verbreitung von Smartphones und Tablets statt dem klassischen Desktop oder Laptop, und Betriebssystemen, insbesondere die zunehmende Verwendung von MacOS[11], tut ein Übriges, um die Verteilung von Software zu erschweren. Damit der eigentliche Lernprozess nicht schon vor dem Start der Entwicklungsumgebung behindert wird, ist eine möglichst einfache Inbetriebnahme von entsprechender Bedeutung. Die Informatiklehrkräfte stehen bei dem Betrieb der jeweiligen Programme in der Schule vor ähnlichen Problemen wie ihre Schüler, nur dass sie zusätzlich auf die Konfiguration des Rechnerpools ihrer Schule oftmals nur einen eingeschränkten Einfluss haben. Die im Vergleich zu privaten Rechnern wesentlich restriktiver gehandhabten Rechte eines Schul-PCs verkomplizieren dieses Szenario zusätzlich. Damit der Lernprozess mit BlattWerkzeug nicht schon mit einer fehlgeschlagenen Installation endet, soll der initiale Kontakt so wenig Hürden wie möglich aufstellen.

Fortführung der entwickelten Projekte

Lernumgebungen wie Scratch oder der AppInventor sind in sich geschlossene Systeme, deren Arbeitsergebnisse nur schwer in anderen Programmen oder Kontexten von Nutzen sind. Sobald der Lernende dann die Grenzen der verwendeten Lernsoftware erreicht hat, steckt er in einem Dilemma: Es wäre notwendig, auf eine andere Software auszuweichen, in diese kann er sein bestehendes Projekt aber nicht einfach mitnehmen. Die Arbeitsergebnisse dieser zu entwickelnden Software

[10] `https://www.geogebra.org/manual/en/Tutorial:Custom_Tools_and_Customizing_the_Toolbar`

[11] `http://de.statista.com/statistik/daten/studie/158102/`

sollen daher zumindest einfach einsehbar sein, optimalerweise nach einem Export sogar mit gängigen Entwicklungsumgebungen oder Texteditoren erweiterbar.

3.3. Künftigen Erweiterungen vorbehalten

Neben der Formulierung von klaren Grundprinzipien als Richtline für relevante Funktionen ist es auch wichtig, den Umfang des zu entwickelnden Programms auf eine im Rahmen der Thesis machbare Teilmenge zu beschränken. Die folgenden Ideen wären naheliegende Ergänzungen, welche aufgrund der zur Verfügung stehenden Zeit aber zunächst nicht implementiert werden sollen[12]. Der Zweck dieser Master-Thesis ist „lediglich“ die Implementierung eines Prototypen. Die Fertigstellung von BlattWerkzeug soll im Rahmen weiterer Arbeiten erfolgen und die im Folgenden geäußerten Ideen werden dann erneut evaluiert.

Datenmodellierung

Der Schwerpunkt dieser Arbeit liegt zunächst auf der Vermittlung von Kenntnissen zur Abfrage und Manipulation von Daten in einem bestehenden Schema. Änderungen an diesem Schema sind nicht vorgesehen, demzufolge ist auch der Neuentwurf eines Schemas mit externen Mitteln zu bewerkstelligen.

Aufwändiges Design von Benutzerschnittstellen

Auch wenn die Konzeption der Benutzerschnittstelle für die verschiedenen Masken in den eben aufgezählten Prinzipien auftaucht, ist es wichtig, den engen Rahmen dieses Aspektes zu verstehen. Es geht um die Schaffung von einfachen, datenzentrierten Eingabemöglichkeiten, nicht um die Umsetzung besonders kreativer optischer Feuerwerke. Dementsprechend ist z.B. die Erweiterung der zur Verfügung stehenden Eingabeelemente durch die Lernenden außerhalb des Rahmens dieser Arbeit. Technisch gesehen schließt dieses Kriterium die Unterstützung von `CSS` oder JavaScript innerhalb von BlattWerkzeug im Rahmen dieser Arbeit aus.

Serverseitige Programmierung mit typischen Programmiersprachen

Auch wenn BlattWerkzeug-Anwendungen grundsätzlich interaktiv sein sollen, ist diese Interaktivität in einem sehr engen Rahmen zu sehen. Für die Darstellung der Oberfläche könnten zwar Kontrollstrukturen wie Schleifen oder Verzweigungen

[12] Die griffigere, englische Bezeichnung dafür wäre „out of scope“, dazu kennt allerdings auch das Internet keine direkte Übersetzung: `http://german.stackexchange.com/questions/31085/`

zum Einsatz kommen, diese operieren aber stets auf einem während des Rendervorgangs unveränderlichen Datenbestand. Die einzige Möglichkeit zur Veränderung von Daten zur Laufzeit ist die Verwendung von mutierenden `SQL`-Anweisungen, klassische Zuweisungen finden nicht statt.

3.4. Allgemeines Konzept der Entwicklungsumgebung

Dieses Kapitel beschreibt das Konzept hinter BlattWerkzeug, ohne dabei jedoch auf die spezifischen Editoren für `SQL` oder Webseiten einzugehen.

Die Oberfläche orientiert sich dabei grundsätzlich an der typischen Dreiteilung einer Entwicklungsumgebung (Abbildung 15). Der Dateibaum bietet jederzeit Zugriff auf alle Inhalte des aktuellen Projekts und ist in der Darstellung unabhängig von dem aktuell zu bearbeitenden Element.

Abbildung 15: Dreispaltiges Layout, ähnlich einer normalen Entwicklungsumgebung.

Der größte Bereich steht den jeweiligen Editoren zur Verfügung. Deren Gestaltung ist dabei Gegenstand eigener Kapitel (3.6 „Drag & Drop-Editor für `SQL`“ & 3.8 „Drag & Drop-Editor für Oberflächen“). Die so genannte „Toolbox“ bietet zu dem aktuell geladenen Editor kontextuell passende Optionen an. Dabei soll es grundsätzlich möglich sein, die in der Toolbox angebotenen Optionen per Drag & Drop im Editorbereich zu verwenden.

Darüber hinaus existieren für jeden Editor spezielle Aktionen, welche über einen Knopfdruck oder auch ein Tastaturkürzel ausgelöst werden können. Diese Aktionen werden oberhalb der Editorfläche in einer typischen „Toolbar“ angezeigt. Einige Aktionen, zum

Beispiel „Speichern“ mit dem Kürzel `STRG` + `S`, stehen dabei in allen editierbaren Kontexten zur Verfügung. Eine Sonderrolle nimmt dabei die Löschfunktion ein: Immer wenn in einem beliebigen Editor eine Drag-Operation beginnt, kann diese auf einem in der Toolbar angezeigten Mülleimer beendet werden.

Grundsätzlich unterschieden wird bei der Schülerentwicklungsumgebung, ähnlich wie bei Scratch, zwischen zwei Blickwinkeln auf das jeweilige Projekt: Zunächst wird ein Projekt im Entwicklermodus editiert, in diesem Fall stehen dem Benutzer alle Entwicklungstools zur Verfügung. Wenn es dann später einmal fertig ist und an Endanwender, z.B. Bekannte oder Freunde, weitergegeben wird, erwarten diese natürlich eine normale Benutzeroberfläche.

Der Wechsel zwischen diesen beiden Modi sollte dabei, ebenfalls analog zu Scratch, zu jedem Zeitpunkt möglich sein. Der Weg von der Entwicklungsumgebung zum Programm ist dabei selbsterklärend: Im Rahmen einer Vorschaufunktion müssen sich die erstellten Seiten jederzeit anzeigen lassen. Aber auch der „Rückweg“ von einer speziellen Seite der Endbenutzeroberfläche zur Entwicklungsumgebung ist inhaltlich sinnvoll. So kann man sich als Lernender die Herangehensweise einer speziellen Seite in einem anderem Projekt anschauen. Das „Verstecken“ von Quelltexten hingegen ist im Lehrbetrieb nicht sinnvoll und daher auch nicht vorgesehen. Jedes fremde Projekt soll auch als Inspiration für die Umsetzung eigener Ideen dienen können.

3.4.1. Webanwendung

Um die einfachste Verwendung der Software für Lernende zu gewährleisten, wird Blatt-Werkzeug als Webanwendung entwickelt. Für die ersten Schritte der Softwareentwicklung mit SQL reicht auf Seite der Lernenden dann ein beliebiger, aktueller Browser. Wie aus Kapitel 2 „Vergleichbare Arbeiten und Produkte“ ersichtlich, befindet sich Blatt-Werkzeug mit diesem Ansatz in guter Gesellschaft: Das „große Vorbild“ Scratch ist über den Browser erreichbar, genau so wie auch die anderen explizit als Lernsoftware beschriebenen Anwendungen.

Selbstverständlich muss eine Entscheidung mit einer solchen Tragweite dennoch gut abgewogen werden. Dass vergleichbare Arbeiten sich ebenfalls für diesen Ansatz entschieden haben ist zwar ein starkes Indiz, isoliert betrachtet aber natürlich keine stichhaltige

Begründung. Immerhin existiert von Scratch auch eine lokal installierbare Fassung, welche der Entwicklung der Webversion allerdings leicht hinterherhinkt[13].

Praktisch bedeutet dieser Ansatz vor allem eine Verschiebung der Probleme mit der Inbetriebnahme auf die Lehrperson. Bei der Konzeption von BlattWerkzeug wird davon ausgegangen, dass die Bereitstellung eines Serverdienstes in Zeiten von Virtualisierungs- und Containerumgebungen für Informatik-affine Lehrkräfte keine nennenswerte Hürde mehr darstellt.

Ebenfalls aus Gründen der einfacheren Zugänglichkeit sollte eine einzelne Serverinstanz in der Lage sein, mehrere Projekte simultan zu bedienen. Die Lehrperson kann also mit einem einzigen Serverprozess eine ganze Klasse versorgen. Eine Begleiterscheinung dieses zentralen Angebots ist allerdings die notwendige Implementierung von zumindest rudimentären Zugriffsbeschränkungen: Im Normalfall sollen Schüler die Projekte ihrer Klassenkameraden zwar jederzeit begutachten, aber nicht modifizieren können. Würde die Entwicklungsumgebung von den Schülern lokal betrieben, entfiele die unmittelbare Notwendigkeit einer Zugriffsbeschränkung.

Durch den Betrieb eines zentralen Servers für die Projekte der Schüler entfällt auch eine weitere typische Problemquelle im Unterrichtsalltag: Sofern der Server tatsächlich durchgehend betrieben wird, ist es sehr einfach auch von Zuhause aus an den Projekten weiter zu arbeiten. Die Notwendigkeit, manuell Dateien über `USB`-Sticks, Netzlaufwerke oder sonstige Dienste zu synchroniseren, entfällt.

Ein weiterer Vorteil eines zentralen Servers ergibt sich in der praktischen Vorzeigbarkeit der eigenen Ergebnisse: Durch einfache Weitergabe der `URL` für Endbenutzer können die Lernenden ihre entwickelten Applikationen mit Eltern, Freunden und anderen Personen teilen.

Nicht verschwiegen werden sollen allerdings auch die negativen Nebeneffekte dieser grundsätzlichen Entscheidung: Zunächst einmal wird dadurch ein eigenständiger Betrieb der Entwicklungsumgebung durch Schüler deutlich erschwert. Diese müssten auf ihrem eigenen Rechner plötzlich doch eine Server-Instanz betreiben, anstatt einfach ein Programm zu starten. Faktisch wird die Bereitstellung einer dedizierten Umgebung für den Server also zur Pflichtaufgabe der jeweiligen Lehrperson. Vom Betrieb eines dedizierten Servers in der Schule über die Verwendung eines „privat greifbaren“ Servers der

[13] Scratch Download-Seite: "The backpack is not yet available."

Lehrperson bis hin zur Verwendung einer Cloud-Computing-Instanz im Internet sind unterschiedlichste Szenarien denkbar. Der erfolgreiche Einsatz von BlattWerkzeug hängt in der Praxis dann aber auch maßgeblich von der Verfügbarkeit dieses Serverdienstes ab.

Softwaretechnisch nachteilig ist die Tatsache, dass BlattWerkzeug im Falle einer Entwicklung als „traditionelle Desktopawendung“ auf ein großes Ökosystem aus bestehenden Entwicklungsumgebungen zurückgreifen könnte. Jede größere Entwicklungsumgebung (Eclipse, Visual Studio, ...) bietet eine Plugin-Schnittstelle, auf die man mit BlattWerkzeug hätte aufsetzen können. Ähnlich gut erprobte Umgebungen für Browser gibt es hingegen leider nicht, es müssen daher relativ viele technische Kleinigkeiten (Toolbars, Projektbrowser, ...) extra für BlattWerkzeug entwickelt werden.

3.4.2. Projektbasiert

Der Begriff des „Projekts“ ist schon einige Male ohne exakte Definition verwendet worden, diese wird daher an dieser Stelle nachgeholt. Abstrakt betrachtet handelt es sich bei einem Projekt um eine Datenbasis in Form einer Datenbank. Zu dieser Datenbasis kann man als Entwickler SQL-Abfragen entwickeln (3.6 „Drag & Drop-Editor für `SQL`“) und einige dieser Abfragen mit einer Oberfläche für Endbenutzer versehen (3.8 „Drag & Drop-Editor für Oberflächen“).

Als unmittelbarer Ausgangspunkt eines Projektes dienen einfache, beliebige SQLite-Datenbanken[14]. Deren Schemata werden vorher mit externen Programmen erzeugt, was sowohl im Dialog mit den Schülern als auch „einfach so“ erfolgt sein kann.

Im Sinne einer möglichst niedrigen Einstiegshürde soll es den Anwendern leicht gemacht werden, bestehende Projekte zu übernehmen. Zu Beginn müssen im Informatikunterricht häufig bestimmte Schritte ausgeführt werden, „weil das nun mal so ist“ (also eine ausführliche Erklärung zu diesem Zeitpunkt zu weit gehen würde). Im Falle des Sprachumfangs von SQL ist das zwar im Vergleich zu z.B. Java nicht ganz so drastisch[15], aber nach Möglichkeit zu vermeiden. Dementsprechend sollte normalerweise jeder Schritt beim Anlegen eines neuen Projektes für die Schüler nachvollziehbar sein. Wenn dann aber doch einmal eine Serie von mechanisch auszuführenden Anweisungen erforderlich sein sollte, wäre es

[14] Kapitel 4.4 „Datenbanksystem“ erläutert, warum ausgerechnet SQLite zum Einsatz kommt und nicht ein anderes Datenbanksystem.

[15] „Herr Lehrer, was macht eigentlich dieses `public static void`?“

dennoch praktisch, diesen Vorgang durch eine Kopie eines schon bestehenden Projektes abzukürzen.

Die Lehrkraft würde in diesem Fall ein Projekt für die Schüler vorbereiten, welches diese dann mit einem Klick in ein anders benanntes Projekt kopieren können. Es werden dann im Normafall alle Bestandteile, möglicherweise mit einer einfachen Anpassung der Zugriffsrechte, unter dem neuen Namen verfügbar gemacht. Alternativ sollte eine Lehrkraft den Umfang eines solchen Klons auch einschränken können: Sofern das Beispielprojekt Inhalte enthält, die nur zu Illustrationszwecken angelegt worden sind, können diese ausgeschlossen werden. Eventuell sollen sich die Schüler auch verstärkt eigene Gedanken machen und nicht zu sehr von vorgefertigen Inhalten abgelenkt werden? Letzten Endes existieren zu viele Varianten, um den korrekten Umfang für diese Kopien ohne manuelle Unterstützung festlegen zu können.

3.4.3. Zugriffskontrollen

Lesende Zugriffe können, zumindest in der Thesis-Version der Software, nicht endgültig eingeschränkt werden. Ursächlich dafür ist jedoch anders als bei vielen anderen nicht implementierten Funktionen nicht vornehmlich der technische Aufwand: Natürlich könnte man theoretisch typische Entwurfsmuster für Zugangsbeschränkungen auch in Blatt-Werkzeug implementieren. Auf einfachster Ebene wäre zum Beispiel denkbar, dass eine Spalte „geheim“ zu Tabellen mit privaten Daten hinzugefügt wird und auf dieser Basis die Oberfläche für Endanwender solcherart markierte Daten nicht anzeigt.

Da Projekte aber auch einfach kopiert werden können, müsste ein versierter Benutzer nur eine Kopie des Projektes vornehmen und könnte dann über den Editor alle Daten einsehen oder auch den Zugriffschutz der Oberflächen einfach deaktivieren. Um also den lesenden Zugriff effektiv zu verhindern, müsste die Einsicht in die Quellen fremder Projekte eingeschränkt werden.

Prinzipiell wäre es mit entsprechendem Zeitaufwand möglich, ein hinreichend sicheres System für den speziellen Anwendungsfall von BlattWerkzeug zu entwerfen. Dieser Aufwand wäre jedoch, im Vergleich zu anderen Features nicht zu rechtfertigen und stünde der Zielstellung „Jedes Projekt soll als Grundlage für neue Projekte dienen können“ obendrein diametral gegenüber. Abgesehen davon sollte kein Schüler einem BlattWerkzeug-Server Daten anvertrauen, deren versehentliche Veröffentlichung er oder sie nicht ver-

kraften würde. Das gilt selbstverständlich auch für den Fall, dass eine hypothetische künftige Version einen solchen Zugriffsschutz vorsehen sollte!

Schon um Vandalismus zu vorzubeugen, sind dennoch einfache Zugriffskontrollen für **schreibende Zugriffe** auf Projektebene nötig. Um etwaige Missverständnisse von vornherein zu vermeiden: Ziel dieser Kontrollen ist es zu verhindern, dass z.B. die Quellen für die Abfragen oder Benutzeroberfläche eines Projektes entstellt oder gelöscht werden. Dabei muss es sich nicht einmal um bösartigen Vandalismus handeln. Bei ungeschickt programmierten Webseiten kann es durchaus passieren, dass Webcrawler Datenbestände vernichten. Mehr als einen schlecht programmierten „Diesen Datensatz Löschen“-Link auf einer Übersichtsseite braucht es dazu nicht. Der Webcrawler folgt unter Umständen auch solchen Verweisen und löscht damit während der Indexierung unbeabsichtigt Datensätze.

Nicht geschützt werden damit einzelne Datensätze, also die Zeilen, in der Datenbank. Sobald in der Benutzeroberfläche für Endanwender Funktionalität zum Löschen oder Editieren von Datensätzen vorgesehen ist, gibt es keine einfache Möglichkeit, legitime Veränderungen von unerwünschtem Vandalismus zu unterscheiden. Daher wird zunächst nur eine sehr einfache, binäre Klassifikation in zwei Benutzergruppen vorgenommen: „Darf jede mutierende Operation ausführen“ oder „darf ausschließlich lesen“. In einer späteren Version wäre es dann denkbar, diese sehr grobe Unterscheidung zu verfeinern.

Sowohl für den lesenden als auch für den schreibenden Zugriff gilt darüber hinaus: Eine komplexe, aber unausgereifte Lösung für eine so kritische Funktionalität würde im Extremfall mehr Schaden anrichten als nützen. Eine unsicher implementierte Sicherheitsfunktion suggeriert möglicherweise ein Schutzniveau, welches praktisch nicht eingehalten wird. Also werden Ansätze wie „Vergabe von Schreibrechten für einzelne Abfragen an bestimmte Benutzergruppen“ oder sogar ein zeilenweiser Schutz von Daten („Jeder Benutzer darf nur seine Daten bearbeiten“) aus Zeitgründen im Rahmen dieser Thesis nicht weiter betrachtet.

Um also zumindest einen grundsätzlichen Schutz zu ermöglichen, sollte für ein Projekt zwischen drei schreibenden Zugriffsarten für angelegte Benutzer gewählt werden können:

Achtung: Nur zur Sicherheit sei an dieser Stelle nochmals deutlich erwähnt, dass dieser Schutz nur für schreibende Zugriffe greift. Der lesende Zugriff auf den gesamten Datenbestand ist über die Entwicklungsumgebung auch für Gäste stets uneingeschränkt möglich.

Entwickler

Mit diesem Zugangslevel hat ein legitimierter Benutzer das Recht, beliebige Änderungen an dem Projekt vorzunehmen. Dabei ist es egal, ob er mit der Entwicklungsumgebung oder der Oberfläche für Endbenutzer arbeitet.

Benutzer

Diese Personengruppe hat keine Möglichkeit, den Quelltext eines Projektes zu editieren, kann aber uneingeschränkt auf die Funktionalität der Oberfläche für Endbenutzer zugreifen. Sofern z.B. das Löschen von bestimmten Datensätzen über die Oberfläche für Endbenutzer möglich ist, können diese Daten von dieser Personengruppe im Rahmen der bereitgestellten Abfragen auch uneingeschränkt gelöscht werden. Wenn hingegen keine mutierenden Operationen über die Oberfläche für Endbenutzer bereitgestellt werden, können auch keine Daten bearbeitet werden.

Gast

Gäste können keinerlei Schreibzugriffe vornehmen, weder über die Entwicklungsumgebung, noch über die Oberfläche für Endbenutzer.

3.4.4. Unterschiedliche Versionen eines Editors

Wenn ein Anwender der Entwicklungsumgebung an die Grenzen des unterstützenden Editors stößt, sollte er die Möglichkeit haben, einmalig für eine konkrete Abfrage oder Seite eine Umwandlung in eine textbasierte Darstellung vornehmen zu können. Der umgekehrte Weg, also zum Beispiel der Import von beliebigen `SQL`-Abfragen in den grafischen Editor, ist dann jedoch nicht mehr möglich. Die Implementierung eines Parsers für die Umwandlung der textuellen Repräsentation in die interne Darstellung von BlattWerkzeug würde im Verhältnis zum marginalen Nutzen einen unverhältnismäßigen Aufwand bedeuten.

Diese unterschiedlich fortgeschrittenen Sichtweisen und Bearbeitungsmöglichkeiten auf jede Ressource, erlauben es BlattWerkzeug auch mit weiter fortgeschrittenem Kenntnisstand effektiv zu nutzen. Dieser Umstand wird vor allem durch die Entscheidung zugunsten von Standardtechnologien anstatt eigens entwickelten Sprachen möglich gemacht. Für Scratch existiert zum Beispiel keine rein textuelle Darstellung mit erweitertem Funktionsumfang. Durch den `SQL`- bzw. `HTML`-Unterbau von BlattWerkzeug stehen fortgeschrittenen Anwendern hingegen sehr mächtige Standardtechnologien zur Verfügung.

3.5. Konzept für sinnvolle Teilmengen von `SQL`

Die uneingeschränkte Umsetzung des `SQL`-Standards würden sowohl für den Verfasser dieser Thesis als auch für die Anwender der Entwicklungsumgebung viele unpraktische Probleme aufwerfen. Der `SQL`-Sprachumfang ist mittlerweile gewaltig und die Anzahl der verschiedenen Dialekte unüberschaubar groß. Varianten wie Transact-`SQL` bieten die Möglichkeit, zur Laufzeit mit Schleifen, Fallunterscheidungen und Variablen zu arbeiten und erfüllen damit alle Kriterien für Turing-Vollständigkeit. Darüber hinaus werden „echte“ Entwicklungsumgebungen für `SQL` von großen Entwicklerteams konzipiert und programmiert, der Autor dieser Arbeit bildet hingegen ein Ein-Mann-Team.

> **Achtung**: Nicht alle hier beschriebenen Optionen sind notwendigerweise in dem Prototypen implementiert! Dieses Kapitel ist eine *allgemeine* Betrachtung von didaktisch sinnvollen Teilmengen von `SQL`. Um zu sehen, welcher Funktionsumfang im Prototypen tatsächlich verfügbar ist, lohnt sich ein Blick in 5 „Fazit“ und den Anhang A „Projektbeispiele“.

Zusätzlich wäre aber auch der Zielgruppe dieser Entwicklungsumgebung mit einer vollständigen Umsetzung des `SQL`-Standards überhaupt nicht gedient. Im Gegensatz zu anderen vorhandenen Entwicklungsumgebungen soll die Stärke dieser Entwicklungsumgebung in einer sinnvollen Reduktion liegen. Dieses Kapitel beschreibt daher, wie man zum „normalen“ `SQL` kompatible Untermengen definieren kann, um einige spezifische Aufgaben zu erfüllen. Oder in Anlehnung an die „80/20“-Regel ausgedrückt: Es gilt, jene 20% des Sprachumfangs abzubilden, mit denen sich 80% der typischen Aufgaben gut lösen lassen. Vor diesem Hintergrund dient dieses Kapitel also ebenfalls als Grundlage für die

Entscheidung bezüglich des Umfangs, mit dem der `SQL`-Editor im Rahmen dieser Arbeit als „abgeschlossen“ betrachtet wird (siehe 3.5.3 „Sprachstufen“).

Bei einer gängigen Programmiersprache wie z.B. Java bauen die einzelnen Sprachfeatures sehr viel stärker aufeinander auf, es ergeben sich deutlich weniger Reihenfolgen, in denen man diese sinnvoll aktivieren könnte. Die wesentlich größere Standardbibliothek einer solchen Sprache trägt dann ebenfalls dazu bei, die Suche nach voneinander weitestgehend unabhängig funktionierenden Teilsprachen zu erschweren.

Glücklicherweise erlaubt die Struktur von `SQL` die Verwendung von sehr lokalen Einschränkungen, deren Auswirkungen sich gut abschätzen lassen. Einzig der Umgang mit Ausdrücken überspannt fast alle Bereiche von `SQL`: Diese können an sehr vielen Stellen auftreten und müssen dementsprechend sorgfältig analysiert werden. Durch diese vergleichsweise überschaubaren Abhängigkeiten ergeben sich viele mögliche Reihenfolgen in denen man Sprachfeatures aktivieren kann.

Es ist nicht im Sinne von BlattWerkzeug, diese unterschiedlichen Reihenfolgen implizit zu bewerten, in dem eine „beste“ Variante fest eingebaut wird. Viel mehr sollen die Lehrkräfte in die Lage versetzt werden, je nach verwendetem Lehrmaterial zu entscheiden, welcher Umfang zum Umgang mit diesem angemessen ist. Dieses Kapitel klärt daher zunächst, welche Features von `SQL` überhaupt für die Entwicklungsumgebung von Belang sind und wie diese voneinander abhängen.

3.5.1. Mögliche Einschränkungen der Komponenten

Die folgende Beschreibung untersucht zunächst nur die Komponenten einer `SELECT`-Suchanfrage. Diese sind im Gegensatz zu `DELETE`, `UPDATE` und `INSERT` nicht auf eine Tabelle beschränkt und auch von der schieren Anzahl der verfügbaren Optionen her deutlich mächtiger.

Zunächst definieren wir einige mögliche Einschränkungen anhand der grundlegenden Struktur jeder suchenden `SQL`-Abfrage. Da die Reihenfolge der Komponenten nicht beliebig ist, ergibt sich eine sehr klare Struktur in der einige Komponenten allerdings auch übersprungen werden können. Abbildung 16 wurde aus den Vorlesungsunterlagen von Prof. Hoffmann entnommen und demonstriert den logischen Ablauf bei der Verarbeitung einer `SQL`-Abfrage inklusive aller Zwischenschritte und optionaler Komponenten. Daraus

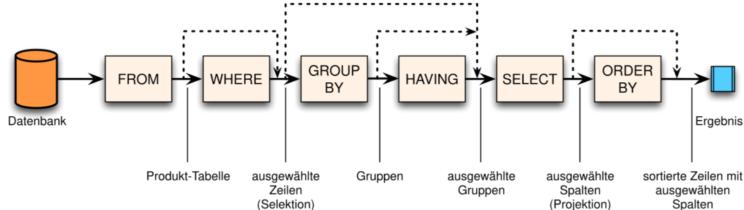


Abbildung 16: `SQL`-Komponenten und sich ergebende Zwischenschritte bei der Verarbeitung einer `SQL`-Abfrage
Quelle: Vorlesung „Einführung in Datenbanken“ von Prof. Ulrich Hoffmann, Folie 19

ist gut ersichtlich, dass alle Komponenten einer Suchanfrage bis auf die logisch Erste (`FROM`) und Vorletzte (`SELECT`) optional sind.

Einige der im Folgenden beschriebenen Optionen sind redundant. Das liegt zum Einen an `SQL` selbst, die Sprache sieht schlichtweg mehrere syntaktische Varianten für semantisch äquivalente Sachverhalte vor. Zum Anderen könnte es aber auch aus didaktischen Gründen sinnvoll sein, die Wissensvermittlung mit einer Reihe von Spezialfällen zu beginnen. Die folgenden Aufzählungen gruppieren die möglichen Optionen für die jeweiligen Komponenten.

1. **Projektionen mit `SELECT`**
   Diese Komponente kann für Suchabfragen nicht ausgeschlossen werden. Außerdem muss mindestens 1a $\vee$ 1b erlaubt sein, sonst können keine gültigen `SQL`-Abfragen erstellt werden.

   a) Auswahl aller Spalten („Sternchen-Operator“ `SELECT *`)
   b) Auswahl von Spalten
   c) Ausdrücke aus einfachen Funktionsaufrufen zulassen
   d) Einfache Ausdrücke zulassen
   e) Beliebige Ausdrücke zulassen
   f) Filterung des Ergebnisses auf unterscheidbare Daten (`DISTINCT`).
   g) Beschränkung der Datenmenge mit `LIMIT`

2. **Angabe von Tabellen mit `FROM`**
   Diese Komponente ist im Normalfall für alle Arten von Abfragen notwendig, obwohl einige `SQL`-Dialekte die Auslassung erlauben. Im Rahmen von BlattWerkzeug wird dies aber nicht unterstützt: Die Schüler sollen `SQL` Abfragen grundsätzlich auf Tabellen beziehen und keine Daten „aus der Luft greifen“[16].

[16]Für Oberflächen werden manchmal noch globale Konstanten benötigt, zum Beispiel der Name des

a) Kreuzprodukt (`JOIN`) zulassen
b) Kreuzprodukt (komma-separierte Schreibweise) zulassen
c) Automatische innere Verknüpfung zulassen (`NATURAL JOIN`)
d) Innere Verknüpfung zulassen (`INNER JOIN`), erfordert 2h $\vee$ 2i
e) Linke äußere Verknüpfung zulassen (`LEFT OUTER JOIN`), erfordert 2h $\vee$ 2i
f) Rechte äußere Verknüpfung zulassen (`RIGHT OUTER JOIN`), erfordert 2h $\vee$ 2i
g) Volle äußere Verknüpfung zulassen (`FULL OUTER JOIN`), erfordert 2h $\vee$ 2i
h) `USING`-Bedingung zulassen
i) `ON`-Bedingung mit einfachen Ausdrücken zulassen
j) `ON`-Bedingung mit beliebigen Ausdrücken zulassen
k) Unterabfragen zulassen

3. **Filterung mit `WHERE` und `HAVING`**

Da an dieser Stelle Ausdrücke zum Einsatz kommen, sind prinzipiell zwei unterschiedliche Ansätze denkbar. Zum einen die Verwendung einer vordefinierten Menge an Vergleichen mit einer einfachen Struktur, diese werden in 3.5.2 „Mögliche Einschränkungen der Ausdrücke“ beschrieben. Dem gegenüber steht der Ansatz, fast beliebige Ausdrücke zu erlauben. Diese sehr mächtige Variante hat zwar die größte Ausdruckskraft, könnte Lernende aber auch mit einem zu großen Freiheitsgrad überfordern.

a) `WHERE` zulassen
b) `HAVING` zulassen
c) Verknüpfungen mit `AND`
d) Verknüpfungen mit `OR`
e) Einfache Ausdrücke zulassen
f) Beliebige Ausdrücke zulassen

4. **Bildung von Gruppen mit `GROUP BY`**

a) `GROUP BY` zulassen
b) Gruppierung mehrerer Spalten zulassen

5. **Sortierung mit `ORDER BY`**

Bei dieser Komponente fehlt bewusst die Möglichkeit anhand eines Spaltenindexes aus dem `SELECT` zu sortieren. Da die gute Lesbarkeit der Abfragen ein erklärtes

Projektes selbst. Diese Werten kommen dann aber nicht aus der `SQL`-Schicht (siehe 3.7.1 „Datenquellen für Webseiten“).

Ziel der Entwicklungsumgebung ist, wurde auf diese redundante Möglichkeit der Spaltenauswahl verzichtet.

a) `ORDER BY` zulassen, die Auswahl der Richtung (`ASC` und `DESC`) ist dann grundsätzlich möglich
b) Sortierung nach im `SELECT` erwähnten Spalten oder Ausdrücken zulassen
c) Sortierung nach beliebigen Spalten zulassen
d) Sortierung anhand eines beliebigen Ausdrucks zulassen

3.5.2. Mögliche Einschränkungen der Ausdrücke

Nach der Betrachtung der mit der unmittelbaren `SQL`-Struktur zusammenhängenden Funktionalität, widmet sich dieses Kapitel den dort verwendeten Ausdrücken. Um eine graduelle Lernkurve sowie eine sinnvolle Unterstützung durch den Editor zu gewährleisten, werden diese in drei Klassen eingeteilt:

Einfache Ausdrücke

Einfache Ausdrücke unterliegen einer festen Struktur, die nur sehr begrenzt die Schachtelung oder Verkettung von zusammengesetzten Ausdrücken zulässt. Ziel dieser Reduktion ist, dass die Auswertungsreihenfolge dieser Ausdrücke auch ohne die Verwendung von Klammern intuitiv eindeutig ist.

Die Bearbeitung dieser Ausdrücke ist nicht notwendigerweise völlig frei, sondern könnte auch hinsichtlich der möglichen Operatoren oder der Struktur eingeschränkt werden.

Komplexe Ausdrücke

Komplexe Ausdrücke lassen sich schachteln bzw. verketten und können mit einem Scratch-ähnlichen Editor für Ausdrücke frei bearbeitet werden. Die korrekte Beachtung von Auswertungsreihenfolgen ist Aufgabe der Lernenden, weswegen diese ggfs. die Ausdrücke auch selber klammern müssen.

Beliebige Ausdrücke

Beliebige Ausdrücke werden textuell notiert und vom `SQL`-Editor nicht weiter geprüft, sondern ungefiltert in die Abfrage eingesetzt. Sofern diese Art von Ausdrücken zugelassen wird, erfolgt also auch keinerlei inhaltliche Einschränkung der zur Verfügung stehenden Möglichkeiten oder eine Syntaxüberprüfung.

Diese Option soll fortgeschrittenen Lernenden erlauben, auch in nicht von BlattWerkzeug unterstützten Szenarien weiterarbeiten zu können. Damit wird verhindert, dass zu große Fortschritte der Lernenden BlattWerkzeug sofort obsolet machen.

Mit dieser Unterscheidung im Hinterkopf, können wir eine Menge an didaktisch sinnvollen Einschränkungen definieren. Ein wesentliches Lernziel ist dabei, dass es sich bei Ausdrücken (nicht nur in `SQL`) um ein allgemeines Konzept handelt, welches prinzipiell in der `SELECT`-Komponente genau so angewandt werden kann wie beim `ORDER BY`. Oder anders ausgedrückt: Wenn sich die Schüler aus didaktischen Gründen zunächst nur mit Ausdrücken im Rahmen der `SELECT`-Komponente beschäftigen sollen, ist das eine sinnvolle, und daher mögliche Einschränkung. Nicht sinnvoll hingegen wäre es, die Anwendung der `LENGTH(X)` Funktion nur in einer der beiden Komponenten zuzulassen. Letzteres wird daher auch nicht unterstützt.

6. **Allgemeine Optionen**
   Diese Optionen betreffen prinzipiell alle Stellen, an denen die entsprechenden Arten von Ausdrücken zugelassen werden.
   a) Ausdrücke als beliebigen Text zulassen. Diese Option erlaubt die Umgehung aller Einschränkungen für einfache Ausdrücke!
   b) An Stelle von Konstanten auch die Verwendung von Platzhaltern zur Bindung von Daten aus der Oberfläche zulassen.
7. **Einfache berechnende Ausdrücke**
   Diese Ausdrücke tauchen normalerweise in `SELECT`- oder `ORDER BY`-Komponenten auf, möglicherweise aber auch als Bestandteil eines einfachen Vergleichs.
   a) Einfacher Funktionsaufruf mit Spalten oder Konstanten als Argument
   b) Einfache Rechenausdrücke mit einer mathematischen Operation ($alter + 1$, $anzahl * preis$)
   c) Konkatenation von Strings[17]
8. **Einfache Vergleichende Ausdrücke**
   Diese „typischen“ Vergleichsausdrücke sollten ausreichen, um einen Großteil der nötigen Vergleiche in `WHERE`, `HAVING` und `ON` Komponenten abzudecken. Die linke

[17] Sofern auch eine Oberfläche entwickelt werden soll, ist eine Konkatenation im Regelfall lieber dort vorzunehmen.

Seite dieser Ausdrücke ist stets eine zur Verfügung stehende Spalte, der Vergleichsoperator und die rechte Seite eine der hier aufgezählten Möglichkeiten.

Mit Vergleich ist dabei nicht nur der =-Operator gemeint, sondern auch die relationen Operatoren $\neq, <, \geq, \leq$ und $>$. Die komplexeren Operatoren `LIKE` und `IN` sowie der ternäre Operator `BETWEEN` werden aber gesondert aufgeführt.

a) Vergleich einer Spalte mit einem konstanten Wert
b) Vergleich einer Spalte gegen `NULL`
c) Vergleich einer Spalte mit einer anderen Spalte
d) Vergleich einer Spalte mit einem einfachen berechnenden Ausdruck (siehe 7)
e) Test einer Spalte mit dem `BETWEEN`-Operator und zwei Konstanten
f) Nutzung des `LIKE`-Operators mit einer Konstanten
g) Nutzung des `IN`-Operators mit einer Liste von Konstanten
h) Negation aller einfachen Ausdrücke

9. **Allgemeine Funktionen (in alphabetischer Reihenfolge)**
In einem Umfeld für Anfänger ist zu erwarten, dass fast jede Funktion (bzw. jede logische Gruppe an Funktionen) einer eigenen Einführung bedarf. Die Lehrkräfte sollen diese daher isoliert zur Verfügung stellen können, ohne Ablenkung durch nicht benötigte Funktionen.

Bei unterschiedlich überladenen Varianten der gleichen Funktion wird in Blatt-Werkzeug immer nur die vollständigste Variante implementiert. Die Schüler müssen daher nicht die implizit bekannten Werte der jeweiligen Funktion auswendig kennen.

a) `ABS(X)`, um den Betrag eines Wertes zu erhalten
b) `COALESCE(X,Y,...)`, um den ersten gültigen Wert der Parameterliste zu erhalten
c) `INSTR(X,Y,Z)`, um das Vorkommen eines Suchwortes in einer Zeichenkette zu erhalten
d) `LENGTH(X)`, um die Länge einer Zeichenkette zu bestimmen
e) `LOWER(X)`, und `UPPER(X)` um die klein bzw. groß geschriebene Variante einer Zeichenkette zu erhalten
f) `PRINTF(FORMAT, ...)`, zur Formatierung von Zeichenketten wie in der gleichnamigen Funktion der `C`-Standardbibliothek.

g) `RANDOM()`, um einen zufälligen Wert zu erhalten
h) `REPLACE(X,Y,Z)`, um Teilbereiche der Zeichenkette zu ersetzen
i) `ROUND(X,Y)`, um kaufmännisch zu runden
j) `SOUNDEX(X)` um einen phonetisch orientierten, unscharfen Vergleich von Zeichenketten durchzuführen.
k) `SUBSTR(X,Y,Z)`, um einen Ausschnitt der Zeichenketten zu extrahieren
l) `TRIM(X)` sowie `LTRIM(X)` und `RTRIM(X)`, um Leerzeichen am Anfang oder Ende eines Strings zu entfernen

10. **Aggregierende Funktionen (in alphabetischer Reihenfolge)**
Die Verfügbarkeit dieser Funktionen hängt von der Verfügbarkeit der `GROUP BY`-Komponente ab (4a).

a) `AVG(X)` bildet das arithmetische Mittel
b) `COUNT(X)` zählt alle Werte, die nicht `NULL` sind
c) `GROUP_CONCAT(X,Y)` konkateniert einzelne Zellen einer Gruppe
d) `MIX(X)` und `MAX(X)` selektieren einen minimalen oder maximalen Wert
e) `SUM(X)` summiert und gibt im Falle von komplett fehlenden Werten `NULL` zurück
f) `TOTAL(X)` summiert und gibt im Falle von komplett fehlenden Werten `0.0` zurück

3.5.3. Sprachstufen

Angelehnt an die bei „Dr. Racket“ umgesetzte Idee der unterschiedlichen Sprachstufen [3] beschreibt dieses Kapitel konzeptionell, wie Lehrkräfte, aufbauend auf den im vorigen Kapitel beschriebenen Funktionalitäten, sinnvolle Untermengen von `SQL` definieren könnten, auf denen die mitgelieferten Beispielprojekte aufbauen. Bei den hier aufgeführten Beispielen handelt es sich keineswegs um feste Vorgaben. Vielmehr soll es für Lehrkräfte möglich sein, fast beliebige Teilmengen von `SQL` für ihre Schüler vorzugeben, nicht nur die im Rahmen dieser Arbeit als sinnvoll erkannten Kombinationen.

Das grundsätzliche Vorgehen um den Sprachumfang einzuschränken, ist dabei mehrstufig. Initial wird auf Projektebene mit einer Whitelist gearbeitet: Jede explizit erlaubte Funktionalität steht im `SQL`-Editor (siehe Kapitel 3.6 „Drag & Drop-Editor für `SQL`“) zur Verfügung. Und sofern eine Funktionalität nicht erlaubt sein sollte, ist sie auch nicht

verfügbar. Der Begriff „Einschränkung“ ist aus der Sicht einer Whitelist also zwar nicht ganz optimal gewählt, beschreibt die stattfindende Reduktion des Umfangs von `SQL` aber akkurat.

Alternativ sollen sich auch wortwörtliche Einschränkungen im Sinne einer Blacklist angeben lassen. Für die in der Regel sehr übersichtlichen Teilsprachen die in dieser Arbeit konzipiert werden, ist diese Variante häufig schlechter lesbar: Man muss den größtmöglichen Umfang des hier beschriebenen Dialekts immer im Hinterkopf haben. Außerdem kommen im Falle einer Erweiterung des Sprachumfangs, z.B. durch ein späteres Update der Entwicklungsumgebung, möglicherweise unerwünschte Möglichkeiten hinzu.

Danach können die Einschränkungen noch für einzelne Abfragen verfeinert werden. Im allgemeinen Fall wird diese Funktionalität wenig nachgefragt sein: Insbesondere bei der Entwicklung von Webanwendungen ist es nicht hilfreich, sich für jede Abfrage in einen anderen Sprachumfang einarbeiten zu müssen. Sofern die Schüler aber bestimmte Aufgaben mit einem didaktisch motiviertem reduzierten (oder erweiterten) Sprachumfang lösen sollen, kommt diese Möglichkeit zum Einsatz. Die hier exemplarisch aufgeführten Feature-Levels orientieren sich an den im Rahmen der Master-Thesis exemplarisch erstellten Datenbeständen und an den Aufgaben aus Informatik-Lehrbüchern [2, 1].

Arbeiten mit einfachen Einschränkungen
: Die Selektion steht in allen betrachteten Schulbüchern am Beginn des Lernprozesses. Verwendet werden dabei durchgängig alle relationalen Operatoren ($<, \leq, =, \neq, \geq, >$), das logische `NICHT`, `UND` sowie `ODER`. Teilweise werden auch schon einfache unscharfe Textsuchen mit dem `LIKE`-Operator behandelt. Syntaxzucker wie `BETWEEN` kommt jedoch nicht vor.

Arbeiten mit Projektionen
: Der Funktionsumfang von diesem Feature-Level orientiert sich an gängigen Tabellenkalkulationsprogrammen wie Microsoft Excel oder Libre Office Calc. Im Vordergrund steht die Verwendung oder Schachtelung von Text-Funktionen wie `substring` oder `find`. Bei den Datensätzen handelt es sich häufig um stark denormalisierte Daten, welche nicht einmal in erster Normalform vorliegen. So werden zum Beispiel Vor- und Zuname in eine einzige Zelle geschrieben und müssen dann in der Projektion wieder voneinander getrennt werden.

Arbeiten mit aggregierten Daten
: Dieser Schritt ist gewissermaßen die logische Fortführung der Projektion. Die Ler-

nenden müssen verstehen nach welchen Kriterien die Gruppierung erfolgen sollte und welche Spalten danach noch ohne Aggregation zur Verfügung stehen.

Arbeiten mit `INNER JOIN`s
Dieses Feature-Level stellt die Verknüpfung vorhandener Datenbestände in den Vordergrund und blendet jegliche Projektion oder Gruppierung vollständig aus. Schüler sollen auf dieser Stufe lernen, sich in einem Datenmodell anhand der vorhandenen Fremdschlüssel-Beziehungen zu orientieren.

Arbeiten mit `OUTER JOIN`s
Auf dieser Stufe sollen die Schüler erkennen, dass durch einen `INNER JOIN` nicht nur neue Kombinationen von Datensätzen erschlossen werden, sondern auch Zeilen wegfallen können.

Für diese Arbeit wird der Umfang auf einfache Einschränkungen und `JOIN`s beschränkt. Diese beiden Stufen erlauben die Navigation in umfangreichen Datenbeständen, also in Schemata mit beliebig vielen Tabellen. Ohne diese Funktionalität wären die sinnvoll verwendbaren Schemata auf jene mit einer einzelnen Tabelle reduziert.

Die verwendete Templatingsprache (siehe 3.7.3 „Die Templatingsprache Liquid“) verfügt ebenfalls über Funktionen zur Projektion von Daten. Um im Rahmen der Thesis keine redundante Funktionalität zu implementieren, wird zunächst darauf verzichtet transformierende Projektionen (konkret also Funktionen) in `SQL` zu unterstützen.

Aggregierte Daten sind aus Sicht der Oberfläche nicht besonders spannend: Es handelt sich weiterhin um eine Struktur aus Zeilen und Spalten. Die Unterstützung von `GROUP BY` stellt aus Sicht des Datenaustausches zwischen Abfragen und Seiten keine Veränderung dar. Aus Zeitgründen wird daher auch diese Funktionalität nicht implementiert um mehr Zeit für die Entwicklung des Oberflächeneditors zu haben. Der Umgang mit `OUTER JOIN`s schlägt sich hingegen möglicherweise auch in der Oberfläche wieder. Plötzlich müssen dort Platzhalter für nicht vorhandene Werte vorgesehen werden.

3.6. Drag & Drop-Editor für `SQL`

Der grafische Editor soll weitestehend analog zu den aus Scratch bekannten Bedienkonzepten funktionieren können. Es kommen also distinkte Bedienelemente für die verschiedenen Komponenten einer `SQL`-Abfrage zum Einsatz, kein reiner Texteditor. Der

komponentenorientierte Editor soll dabei nicht die Konzeption von beliebigen Abfragen ermöglichen, wohl aber umfassend den Anforderungen aus Schulbüchern wie [1, 2] gerecht werden.

Grundsätzlich bedürfen einige Komponenten der Abfrage besonderer Aufmerksamkeit, weil sie große Auswirkungen auf das Verhalten der anderer Komponenten haben. Vorrangig ist hier die `GROUP-BY`-Komponente zu nennen. Sobald die Query mit einer `GROUP BY`-Komponente ausgestattet wird, ist der Zugriff auf die konkreten Spalten einzelner Zeilen z.B. im allgemeinen Fall nicht mehr möglich. Das Hinzufügen (oder Entfernen) dieser Komponente hat also große Auswirkungen auf die Korrektheit der gesamten Abfrage. Ähnliches gilt für den Umgang mit `JOIN`s: Aus diesen leitet sich ab, welche Daten von der Entwicklungsumgebung überhaupt angeboten werden sollten.

3.6.1. Visuelle Gestaltung

Für technische Details irrelevant, aber unbedingt ebenfalls im Voraus zu klären ist die Frage inwiefern es sinnvoll wäre, das sehr bunte, kontrastreiche Design von Scratch zu imitieren. Abbildung 18 zeigt einen Vergleich zweier Prototypen aus der Konzeptionsphase von BlattWerkzeug: Einmal sehr bunt und „blockig“ und einmal angelehnt an konventionelles Syntax-Highlighting.

Das blockige Design orientiert sich an Scratch und macht visuell sehr deutlich, welche Komponenten logisch zusammenhängen. Im Laufe der Entwicklung des Prototypen wurde aber immer deutlicher, dass die Zusammenhänge der Komponenten hier eher überbetont werden. In Scratch ist dieser deutliche visuelle Zusammenhang notwendig. So zeigen z.B. die Konnektoren der Blöcke für Ereignisse oder Endlosschleifen sehr deutlich, wie sich der Kontrollfluss durch diese Elemente verändern wird: Der Kontrollfluss beginnt mit dem Ereignis, ein Anhängen von Blöcken nach der Schleife ist weder möglich noch sinnvoll (Abbildung 17). Für eine vollständige Programmiersprache ist das mit Sicherheit eine gute Wahl, für den sehr linearen Ablauf einer `SQL`-Abfrage ist diese so ausgeprägte visuelle Hierarchie im Regelfall nicht nötig.

Abbildung 17: Kontrollfluss in Scratch

Die in den Prototypen noch zu sehende Nutzung von „normalen“ Bedienelementen wie Text- oder Listboxen (Abbildung 18 & 19) wurde schon früh in der Entwicklung zugunsten eines auf Drag & Drop aufbauenden Konzeptes aufgegeben. Die Bedienelemente haben zum einen schon rein visuell die Lesbarkeit merklich gestört und hätten häufig ausgeblendet werden müssen. Zudem ist bei der Bearbeitung von Ausdrücken eine distinkte visuelle Hierarchie sehr von Vorteil. Diese Hierarchie lässt sich am einfachsten durch die konsequente Verwendung von klar codierten Formen und Farben erreichen, was sich für vom Browser bereitgestellte Bedienelemente aber nur in engen Grenzen umsetzen lässt[18].

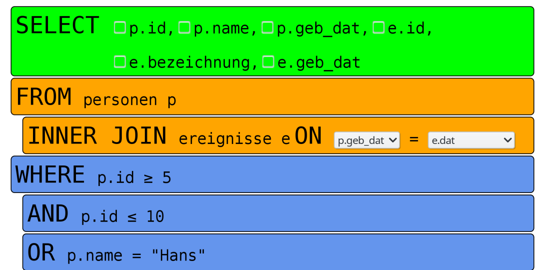
Darüber hinaus existiert noch ein weiterer Faktor, der zwar keinen Einfluss auf die technischen Details von BlattWerkzeug hat, aber trotzdem eine etwas ausführlichere Betrachung rechtfertigt: die größere Nähe des etwas nüchterneren, IDE-ähnlichen Designs zu „normalen“ Entwicklerprogrammen. Diese wirken möglicherweise weniger einschüchternd, wenn man sich schon an den Anblick von recht viel Text mit Syntax-Highlighting gewöhnt hat. Es ändert sich dann bei der Verwendung von typischen Entwicklerprogrammen dann „lediglich“ das Bedienparadigma sehr stark (Text schreiben statt Drag & Drop), aber nicht mehr der visuelle Ersteindruck.

Letztendlich existiert aber auch noch ein sehr viel profanerer Faktor: Der Autor dieser Arbeit ist kein Grafikdesigner und würde ohne Hilfe vermutlich kein ansprechendes, durchgehend „blockiges“ Gesamtkonzept auf die Beine stellen können. Das endgültige Design lehnt sich bei dem Editor für Ausdrücke durch die Verwendung von Blöcken visuell an Scratch an, nutzt zur Visualisierung der Komponenten jedoch eine an Syntax-Highlighting orientierte Darstellung.

3.6.2. Grundsätzlicher Aufbau

Es stellt sich zunächst die Frage, ob der Editor, wie Scratch, mit einer Drag & Drop-Seitenleiste ausgestattet werden sollte oder ob auch ein anderes Bedienkonzept denkbar wäre. Ein grundsätzlicher Nachteil des Drag & Drop Bedienkonzeptes ist der nötige Platz für die Unterbringung aller verwendbaren Komponenten. Spätestens wenn für die Anzeige aller verfügbaren Optionen nicht ausreichend Platz auf dem Bildschirm vorhanden

[18] Historisch waren diese Bedienelemente überhaupt nicht mit `CSS` zu gestalten, diese Situation hat sich heutzutage deutlich verändert. Der nötige Aufwand dafür ist jedoch groß und erfordert zum Beispiel das deaktivieren der nativen Bedienelemente mit `display:none` und dann einen „Nachbau“ mit anderen Elementen.



(a) Starke Betonung der Blöcke

```
SELECT p.id, p.name, p.geb_dat, e.id,
       e.bezeichnung, e.geb_dat
FROM person p
 INNER JOIN ereignisse e ON p.geb_dat = e.dat
WHERE p.id ≥ 5
 AND p.id ≤ 10
 OR p.name = "Hans"
```

(b) Syntax-Highlighting

Abbildung 18: Vergleich unterschiedlicher Gestaltungsansätze

ist, muss sorgfältig geplant werden, wie die entsprechenden Bedienelemente anzuordnen sind.

Scratch nutzt eine Einordnung der Blöcke in zehn Kategorien. Es sind daher nie alle existierenden Blöcke auf einmal sichtbar, sondern immer die thematisch verwandten Blöcke der aktuell gewählten Kategorie. Eine solche Kategorisierung muss sehr sorgfältig entwickelt und vor allem erprobt werden: Sollten die Zuordnung eines Blocks mehrdeutig oder missverständlich sein, besteht die Gefahr, dass er nicht gefunden wird. Im Rahmen dieser Arbeit wurde diese Problematik der Kategorisierung daher erst einmal verschoben: Es wird darauf verzichtet, solange die zur Verfügung stehenden Bedienelemente noch bequem Platz in einem Full-HD-Browserfenster haben.

Da der Benutzer immer nur eine Abfrage zur Zeit bearbeiten können soll, ergibt sich der einzig mögliche Platz für viele Blöcke automatisch. Darüber hinaus ist die Reihenfolge eines Großteils der Komponenten einer Abfrage sehr strikt festgelegt, so kann eine `GROUP BY` Komponente nicht an beliebigen Stellen verwendet werden, sondern nur nach der `FROM` oder der `WHERE` Anweisung. Andere Komponenten wie z.B. `HAVING` oder logische Verknüpfungen mit `AND` oder `OR` sind nicht nur von der Reihenfolge, sondern auch von der Existenz anderer Bestandteile abhängig. Ein möglicher Ansatz zur Reduktion der verfügbaren Komponenten wäre also, für die möglichen Optionen visuell abgegrenzt Platzhalter innerhalb des Abfrageeditors anzubieten. Ein Klick auf den Platzhalter könnte diesen dann in einen konkreten Block umwandeln und gegebenenfalls zur Angabe der benötigten Parameter auffordern.

Im Rahmen der vorab entwickelten, rein visuellen Prototypen hat sich herausgestellt, dass eine permanente Anzeige aller Editierungsmöglichkeiten mit einem sehr überladen wirkenden Benutzerinterface einher geht, insbesondere was die Einblendung von

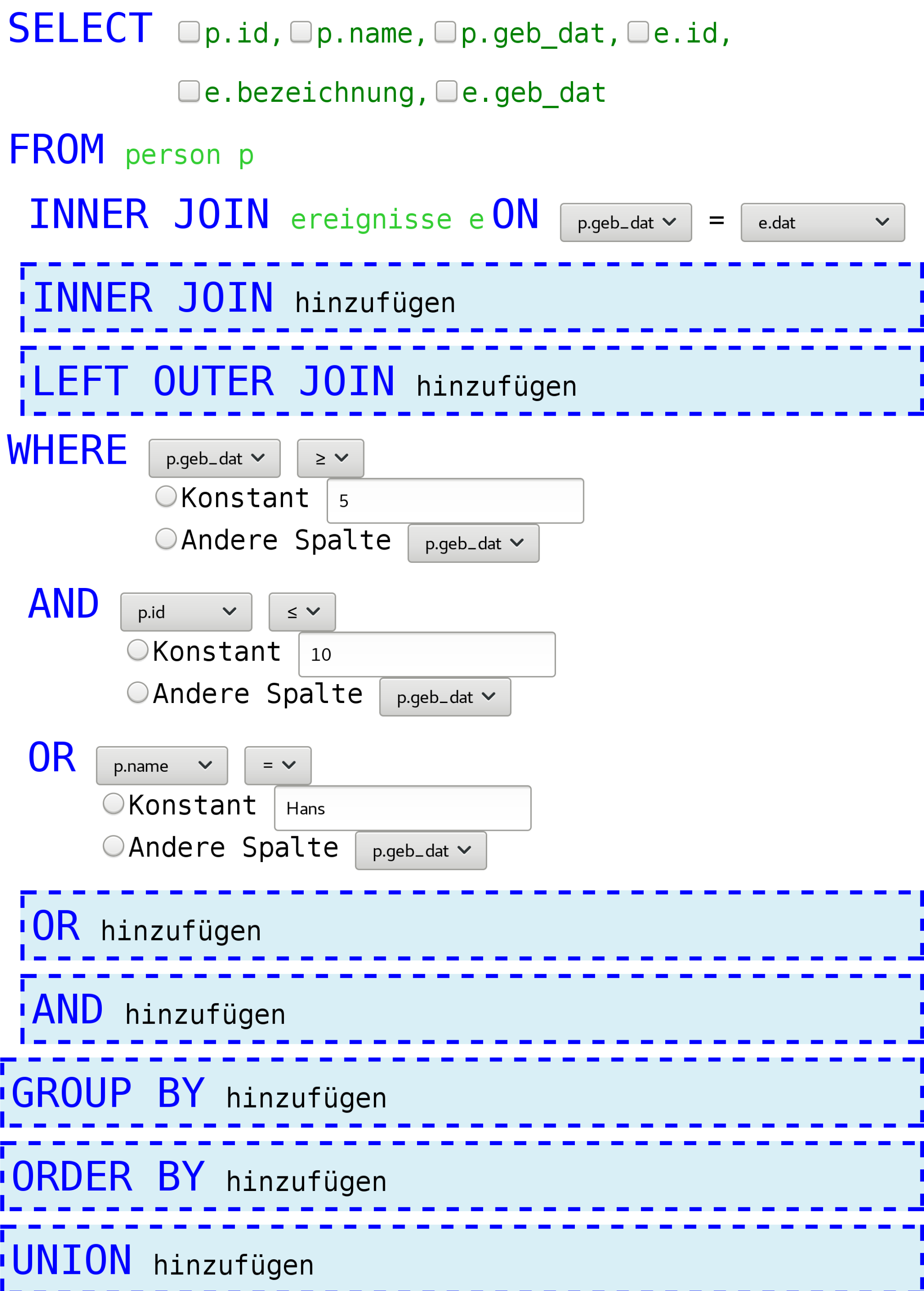


Abbildung 19: Verworfen: Keine Nutzung von Drag & Drop und daher simultane Anzeige (fast) aller Möglichkeiten, zudem ausgiebige Nutzung von „normalen“ Bedienelementen.

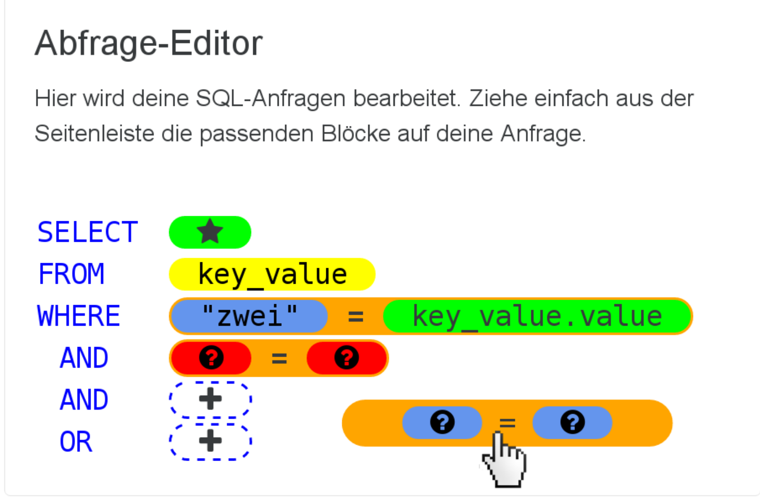
Platzhaltern angeht. Abbildung 19 zeigt einen Screenshot eines Prototypen, bei dem nahezu alle denkbaren Editierungsoptionen gleichzeitig für den Lernenden verfügbar sind und daher angezeigt werden müssen. Trotz der klaren visuellen Unterscheidung fällt es schwer, mit nur einem Blick zu erkennen, was die Abfrage jetzt tatsächlich beinhaltet.

Statt jederzeit alle Optionen anzuzeigen, könnte dann noch die Position des Mauszeigers herangezogen werden, um zu entscheiden, welche Platzhalter angezeigt werden sollen. So wurden alle Editierfunktionen ausgeblendet, wenn sich die Maus außerhalb der Abfrage befand, die `JOIN`-Optionen nur gezeigt, wenn der Cursor sich gerade in der Nähe befand, ... Das Ergebnis war allerdings eine Benutzerschnittstelle, deren Bedienelemente häufig und für einen ungeübten Benutzer auch unerwartet ihre Position verändert haben. Dementsprechend wurde dieser Ansatz noch in der Konzeptionsphase verworfen. Darüber hinaus hätte man mit diesem Ansatz nur hierarchisch verwandte Bereiche ausblenden können: Für die `GROUP BY`-, `ORDER BY`- oder `UNION`-Platzhalter ergibt sich keine intuitive Position des Mauszeigers, bei der diese kontextsensitiv eingeblendet werden könnten.

Das endgültig implementierte Design (Abbildung 20) lehnt sich nun doch an Scratch an, versucht allerdings mit Platzhaltern den Benutzer kontextsensitiv zu unterstützen. Wenn dieser zum Beispiel einen Vergleichsoperator für Ausdrücke in der Seitenleiste anwählt, werden im Hauptfenster die validen Drop-Ziele hervorgehoben. Unter dem eigentlichen Editor befinden sich dann zwei weitere Bereiche (Abbildung 21): Zum einen die Ergebnisanzeige, welche die zur aktuellen Gestalt der Abfrage passenden Zeilen zeigt, zum anderen eine Möglichkeit, bei Abfragen mit Parametern die nötigen Werte zu definieren. An dieser Abbildung wird zudem ein weiteres Detail deutlich: Abfragen lassen sich nur als „einzeilig“ deklarieren, wenn eine `WHERE`-Komponente vorhanden ist.

3.6.3. Umgang mit Ausdrücken

Innerhalb der `SELECT`- und `WHERE`-Komponenten können fast beliebige Ausdrücke auftreten. Diese unterscheiden sich strukturell vor allem anhand ihrer benötigten Parameter. Konstanten, Spalten und „Benutzerwerte“ stehen für sich, können also unmittelbar verwendet werden. Sofern Ausdrücke andere Ausdrücke zur Auswertung benötigen, zum Beispiel das logische `UND`, werden diese zunächst durch einen speziellen Blanko-Platzhalter gefüllt. Auf diesen Platzhalter, symbolisiert durch ein Fragezeichen, lassen sich dann wie



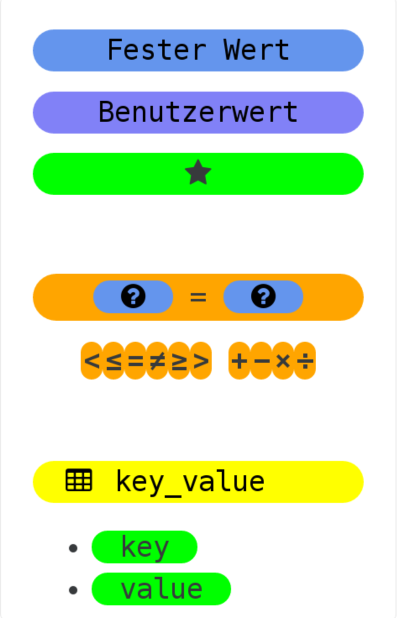


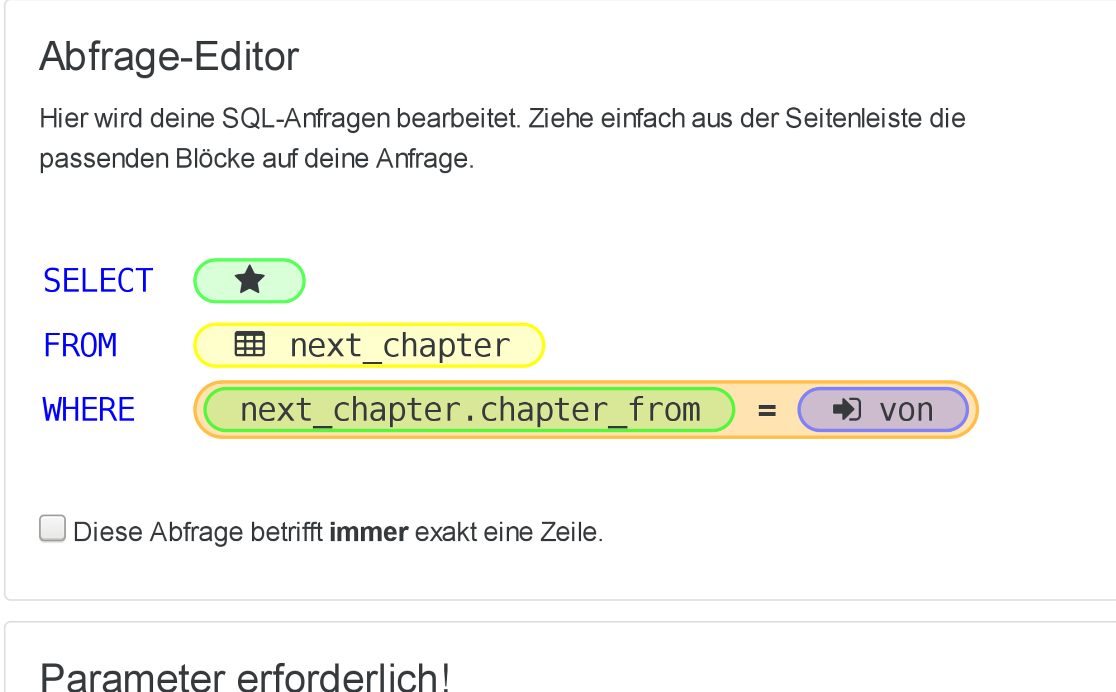
Abbildung 20: Implementiert: Drag & Drop-Interface mit Hervorhebung von möglichen Zielen beim Ziehen aus der Seitenleiste, keine Nutzung von „normalen“ Bedienelementen.

gewohnt per Drag & Drop andere Ausdrücke der Abfrage oder Elemente aus der Seitenleiste ziehen. Damit funktioniert die Entwicklung von Ausdrücken sehr ähnlich, wie es Schülern möglicherweise schon aus Scratch bekannt ist.

Die in Kapitel 3.5.2 „Mögliche Einschränkungen der Ausdrücke“ vorgestellten „einfachen Ausdrücke“ lassen sich schwerer als zunächst vermutet in die Oberfläche integrieren. Schon die Platzierung dieser Vereinfachungen wirft Fragen auf: In der Seitenleiste, einem neuen Dialog oder „irgendwie kontextabhängig“? Sollen für komplexe Sachverhalte eigene „Untereditoren“ entwickelt werden? Wenn es aus didaktischer Sicht hilfreich scheint mit einem „Wizard“-Dialog zu arbeiten, soll dafür das Drag & Drop Paradigma durchbrochen werden? Da sich mit dem Drag & Drop-Ansatz alle Ausdrücke entwerfen lassen, wurde die Diskussion über eine geeignete Benutzerschnittstelle für diese Funktionalität auf unbestimmte Zeit verschoben.

3.6.4. Abfragen mit Parametern

Um eine Interaktion mit Endbenutzern zu ermöglichen, können in Ausdrücken auch benannte Parameter („Benutzerwerte“) anstelle von konstanten Werten verwendet werden. Im Gegensatz zu Konstanten werden diese zur Laufzeit vom Benutzer angegeben. Aus



Parameter erforderlich!

Deine Abfrage verwendet Parameter und funktioniert daher nicht ohne Eingaben vom Benutzer. Wenn du sie trotzdem gerne ausprobieren möchtest, musst du hier testweise Werte eingeben.

von 1

Ergebnisse

Hier kannst du das Ergebnis der Abfrage sehen.

next_chapter.chapter_from	next_chapter.chapter_to	next_chapter.decision_text
1	1	Alles bleibt wie es ist! Du bleibst einfach stehen.
1	3	Du drückst den roten Knopf!
1	5	Türen sind zum Öffnen da.
1	6	Schilder sind spannend! Mal schauen, was darauf steht?

Abbildung 21: Abfrageeditor: Parameter für eine parametriesierte Abfrage und Ergebnisvorschau

Sicht der SQL-Abfrage ist diese Ergänzung trivial, da benannte Parameter ein Bestandteil faktisch jeder SQL-Implementierung. Die eigentliche Problematik liegt im Binden dieser Parameter zur Laufzeit. Das Kapitel 3.7.7 „Bindung von Abfragen an die Oberfläche“ beschreibt, wie die Verknüpfung mit den Bedienelementen der Oberfläche vorgenommen wird.

Innerhalb des Abfrageeditors können solche parametrisierten Abfragen dann folglich nicht mehr direkt ausgeführt werden. Auf diesen Umstand wird der Entwickler, wie in Abbildung 21 zu sehen ist, direkt innerhalb des Editors hingewiesen. Sofern er die benötigten Parameter dann eingibt, kann die Abfrage auch wieder ausgeführt werden.

3.6.5. FROM

Wenn der Benutzer eine Drag-Operation mit einer Tabelle beginnt, kann er die Tabelle wahlweise auf einer bestehenden Tabelle „fallen lassen“, einen ⊕-Indikator nach der letzten Tabelle oder den FROM-Text selbst fallen lassen. Eingefügt wird dabei im Falle von schon existierenden Elementen immer nach dem Ziel. Die ⊕-Indikatoren werden hingegen einfach durch das neue Element ersetzt.

Diese Komponente beinflusst unmittelbar die in der Seitenleiste zur Verfügung stehenden Daten. Für Tabellen, die bisher nicht Teil der FROM-Komponente sind, stehen die entsprechenden Spalten nicht zur Verfügung. Erst wenn man diese Tabelle mit einem JOIN in die Abfrage einbindet, erscheinen auch die dazugehörigen Spalten in der Seitenleiste.

3.6.6. WHERE

Die WHERE-Komponente wird ausgeblendet, sofern der Entwickler noch keine Einschränkungen für die Abfrage definiert hat. Existierende Einschränkungen werden natürlich jederzeit eingeblendet und können durch einfaches „fallen lassen“ anderer Ausdrücke auf die bestehenden Ausdrücke editiert werden.

Erstmalig angelegt (oder mit AND und OR erweitert) wird die Komponente sobald der Entwickler einen Ziehvorgang mit einem Ausdruck beginnt. Im Falle von schon existierenden Einschränkungen werden die AND- und OR-Operatoren angezeigt, andernfalls das

`WHERE` selbst. Hinter den Schlüsselworten wird dabei jeweils mit einem ⊕-Indikator angezeigt, dass dort das aktuell gezogene Element eingesetzt werden kann. Die Ergänzung einer bestehenden Einschränkung wird in Abbildung 20 illustriert.

Syntaktisch gesehen wären die logischen Verknüpfungen natürlich auch als Teil der „normalen“ binären Ausdrücke möglich; in der Praxis wird der Quelltext von `SQL`-Abfragen allerdings häufig anhand dieser Verknüpfungen die Zeile umgebrochen. An dieser Stelle folgt BlattWerkzeug also einer bewährten Konvention, auch wenn das syntaktisch nicht unbedingt notwendig wäre.

3.6.7. `GROUP BY`

Sobald in einer Abfrage eine `GROUP BY` Komponente genutzt wird, hat dies Auswirkungen auf die Möglichkeiten innerhalb der `SELECT` Anweisung. Da keine Auswahl von ungruppierten und nicht-aggregierten Spalten möglich sein darf, müssen diese entfernt werden. Insbesondere wenn der Benutzer vorher schon mit komplizierten Ausdrücken im `SELECT` gearbeitet hat, ist eine Warnung nötig.

3.6.8. Manipulation von Daten

Neben den bisher ausführlich besprochenen `SELECT`-Anweisungen müssen sich natürlich `INSERT`-, `UPDATE`- und `DELETE`-Anweisungen umsetzen lassen. Da diese ihrer Natur nach nicht idempotent sind, verbietet sich natürlich eine wiederholte, automatische Ausführung während der Entwicklungszeit. Es ist zu erwarten, dass die meisten von den Schülern verfassten Abfragen dieser Art ausführlichen Gebrauch von den in Kapitel 3.6.4 „Abfragen mit Parametern“ beschriebenen Parametern machen.

Die Ausführung dieser Operation sollte daher immer in zwei Schritten erfolgen, die von einer Vorschau auf das Ergebnis begleitet werden. Erst wenn diese Ergebnisvorschau vom Entwickler bestätigt wird, werden die Änderungen tatsächlich übernommen. Für `DELETE`- und `INSERT`-Operationen kann diese Vorschau einfach den kompletten neuen bzw. zu löschenden Datensatz anzeigen. Im Falle von `UPDATE`-Anweisungen sollten alle veränderten Datensätze einander gegenübergestellt werden.

Abfrage-Editor

Hier wird deine SQL-Anfragen bearbeitet. Ziehe einfach aus der Seitenleiste die passenden Blöcke auf deine Anfrage.

Aktiv?	Typ	Spalte	Neuer Wert
☑	INTEGER	chapter_from	von
☑	INTEGER	chapter_to	nach
☑	TEXT	decision_text	text

Abbildung 22: Abfrageeditor: Anderes Erscheinungsbild für Spalte-Wert-Paare, wie sie bei `INSRT` oder `UPDATE` Operationen nötig sind

Für die Name-Wert-Paare in `INSERT`- oder `UDPATE`-Operationen löst sich das Erscheinungsbild des Editors von dem Vorbild der `SQL`-Syntax (Abbildung 22). Zur Erinnerung werden an dieser Stelle noch einmal die SQLite-Syntax für `UPDATE`- und `INSERT`-Anweisungen illustriert (Listing 5 & 6). Leider haben diese beiden verwandten Anweisungen offensichtlich eine unterschiedliche Syntax für die Notation der Schlüssel-Wert-Paare.[19]. Dieses syntaktische Detail wird allerdings vor dem Entwickler durch die Oberfläche versteckt. Sichtbar ist dort nur die abstrakte Zuweisung von Werten an eine Auswahl von Spalten.

Um zwischen bewusst und versehentlich nicht vom Entwickler gesetzten Werten zu unterscheiden, ist in der aktuellen Fassung leider noch die Nutzung einer Checkbox vorgesehen. Das ist ein Bruch mit dem üblicherweise verwendeten Drag & Drop-Paradigma

[19]Dieser Umstand wurde mit SQLite Version 3.15.0 für die `UPDATE`-Anweisung verbessert, bisher hat sich diese Syntax substanziell von der `INSERT`-Schreibweise unterschieden. Jetzt ist es analog zum `INSERT` ebenfalls möglich, auf beiden Seiten der Zuweisung identisch lange Listen von Spalten und Werten anzugeben. Allerdings ist Version 3.15.0 am 14. Oktober 2016 erschienen, etwa zwei Wochen vor dem Druck dieser Thesis.

```sql
INSERT INTO gefangen (pokedex_nummer, spitzname, staerke)
VALUES (:nummer, :name, :staerke)
```

Listing 5: Syntax einer `INSERT`-Anweisung

```sql
UPDATE gefangen
SET spitzname = :neuer_name, staerke = :neue_staerke
WHERE gefangen_id = :geaendertes_pkmn;
```

Listing 6: Syntax einer `UPDATE`-Anweisung

und sollte in einer künftigen Version durch aus der Seitenleiste ziehbare Spalten abgelöst werden.

3.7. Konzept für Oberflächen

Damit sich mit der Schülerentwicklungsumgebung erstellte Projekte auch von normalen Endanwendern bedienen lassen, bedarf es natürlich noch einer entsprechenden Benutzeroberfläche. Hier bietet sich, aus Gründen der einfachen Weitergabe, eine webbasierte Oberfläche an. Dadurch entfällt bei den Endanwendern jegliche Installation und gerade für datenzentrierte, verteilte Anwendungen ist eine einheitliche Sicht auf den Datenbestand von großer Bedeutung. Ein weiterer positiver Aspekt ist die einfache Weitergabe der eigenen Arbeitsergebnisse in Form einer `URL`: Diese können über eine beliebige soziale Plattform oder durch einen Messenger im Bekanntenkreis geteilt werden.

> **Achtung**: Nicht alle hier beschriebenen Optionen sind notwendigerweise in dem Prototypen implementiert! Dieses Kapitel ist eine *allgemeine* Betrachtung eines didaktisch sinnvollen Konzeptes zur Beschreibung von Oberflächen. Um zu sehen, welcher Funktionsumfang im Prototypen tatsächlich verfügbar ist, lohnt sich ein Blick in 5 „Fazit“ und den Anhang A „Projektbeispiele“.

Auch wenn im bisherigen Verlauf der Arbeit stets ausschließlich `HTML` als Technologie zur Beschreibung von Oberflächen erwähnt wurde, handelt es sich dabei nur um den statischen Teil des technischen Fundaments. Um die interaktive Anbindung von Datenquellen zu ermöglichen, müssen noch einige Erweiterungen in Form einer Templatingengine zum Einsatz kommen. Ähnlich wie für die `SQL`-Abfragen wird auch für die Oberflächen ein

dedizierter Editor (Kapitel 3.8.4) implementiert. Die Schüler sollten zu keinem Zeitpunkt selber Quelltexte tippen müssen, optional aber durchaus die Möglichkeit dazu haben.

Um das folgende Kapitel anschaulicher zu gestalten, werden die aufgeführten Beispiele meist anhand eines sehr reduzierten Datenbestandes (siehe Tabelle 1 & 2) zu einem einfachen Blog illustriert. Dieser besteht aus zwei Tabellen für Artikel und Kommentare, ein Artikel kann dabei mehrfach oder auch überhaupt nicht über Kommentare verfügen. Bei den Zeitangaben handelt es sich um `UNIX`-Zeitstempel.

article_id	caption	text	date
1	Erster Artikel	Mit wenig Text steht.	1393423200
2	Nächster Artikel	Mit ein bisschen mehr Text.	1456498800
3	Dritter Artikel	Mit `<abbr>`HTML`</abbr>`	1477872000

Tabelle 1: `article`-Daten des Blog-Beispiels

comment_id	article_id	commenter_name	text	date
1	1	Ash Ketchum	Erster!	1393423241
2	1	Betreiber	Zweiter :(	1393435123
3	3	Kommentator	`<i>`HTML`</i>`	0

Tabelle 2: `comment`-Daten des Blog-Beispiels

Dieser Blog soll dem Endbenutzer über zwei Seiten näher gebracht werden: Eine *Hauptseite* dient der Auflistung aller Blogbeiträge, welche im Einzelnen auf einer *Detailseite* in Gänze betrachtet und kommentiert werden können. Dabei können auf der Hauptseite aus Platzgründen nicht die gleichen Informationen angezeigt werden, wie auf den Detailseiten. Daher werden die Daten auf der Hauptseite teilweise aggregiert: Statt alle Kommentare zu einem Beitrag anzuzeigen, wird nur auf die Anzahl der getätigten Kommentare verwiesen.

Bei dieser Aufteilung handelt es sich um eine typische „Master-Detail"-Struktur: Ausgehend von einer sehr komprimierten Darstellung aller Werte können Details zu einem konkreten Datensatz nach einem Navigationsvorgang eingesehen werden. Dieses Navigationskonzept wird von fast allen geschäftlichen Applikationen genutzt und ist eine der Standardvorlagen des Visual Studio Lightswitch.

Abstrakt betrachtet besteht eine BlattWerkzeug-Seite aus zwei wesentlichen Informationen: Einer Auflistung aller zum Darstellen benötigten Datenquellen - im Rahmen dieser Thesis werden das ausschließlich `SQL`-Abfragen sein - und einer Vorschrift, wie

genau dieser Datenbestand darzustellen ist, also eine `HTML`-Vorlage mit speziellen Anweisungen zur Datenanbindung. Kapitel 3.7.1 „Datenquellen für Webseiten“ erläuert, wie diese Anbindung allgemein organisiert wird, Kapitel 3.7.7 „Bindung von Abfragen an die Oberfläche“ beschreibt die konkreten Details der `SQL`-Anbindung.

3.7.1. Datenquellen für Webseiten

Um die einer Webseite zur Verfügung stehenden Daten zu strukturieren, wird der verfügbare Datenbestand in unterschiedliche Namensräume eingeteilt. Dadurch werden Kollisionen von Bezeichnern vermieden und es ist schon anhand der Bezeichnung eindeutig, aus welchem Kontext ein abgerufener Wert zur Verfügung gestellt wird.

`page` für Eigenschaften der Seite
: Der Name der aktuellen Seite ist für die Darstellung möglicherweise relevant und sollte daher während des Rendervorgangs zur Verfügung stehen.

`project` für Eigenschaften des Projektes
: Wenn man als Entwickler globale Konstanten ablegen möchte, könnte man dies auch in der Datenbank tun. Einfacher ist aber ein separater Speicherort, welcher tatsächlich keine Modifikationen durch Endbenutzer vorsieht. Das typische Beispiel für einen solchen global verfügbaren Wert wäre der Name des Projekts an sich oder die Routing-Tabelle mit allen verfügbaren Seiten.

`get` für `URL`-Parameter
: Wenn aus einer Vielzahl von verfügbaren Daten zu einem bestimmten Datensatz navigiert werden soll, sieht der `HTML`-Standard die Verwendung von `URL`-Parametern vor. Mit diesen lassen sich applikationsspezifische Merkmale der anzuzeigenden Datensätze angeben. Im Falle des Blog-Beispiels wäre das zum Beispiel die `article_id` zur Anzeige eines bestimmten Beitrages oder die Angabe eines Startzeitpunktes, um die Hauptseite in mehrere Seiten für unterschiedliche Zeiträume zu unterteilen.

`query` für `SELECT`-Abfragen
: Nicht jede Webseite benötigt die Ergebnisse aller verfügbaren Abfragen. Daher werden die tatsächlich benötigten referenzierten Abfragen in diesem Namensraum gesammelt. Kapitel 3.7.7 „Bindung von Abfragen an die Oberfläche“ beschreibt im Detail, wie diese Anbindung funktioniert.

`form` für Benutzereingaben
: Dieser Namensraum hat im Gegensatz zu allen anderen Namensräumen keinen globalen Gültigkeitsbereich, sondern hängt eng mit dem `HTML`-`form`-Element zusammen. Die zur Verfügung gestellten Daten hängen folglich von der Struktur des Templates ab.

Die Verwendung von Namensräumen ist von Jekyll inspiriert, `page` und `project` findet man auch dort. Durch die Kombination der in diesen Namensräumen verfügbaren Namen für Projekt und Seite kann BlattWerkzeug automatisch eine sinnvolle Implementierung für den `HTML`-Titel jeder Seite generieren: „`page.name - project.name`". Zur Laufzeit würde dann für eine denkbare „Impressum"-Seite des „Beispielblog" also „Impressum - Beispielblog" als Titel angezeigt werden.

Perspektivisch interessant wäre noch die Einbindung von Namensräumen für Cookies und Sessions, um während eines Besuchs (oder auch darüber hinaus) Informationen ablegen zu können. Diese beiden Datenspeicher entfalten ihr volles Potenzial allerdings erst, wenn ihnen dynamisch Werte zugewiesen werden können. Natürlich wäre es nicht undenkbar, dass BlattWerkzeug auch hier eine spezielle Lösung zur Verfügung stellt. Sinnvoller erscheint aber die Verwendung einer „normalen" Programmiersprache, was außerhalb des abgesteckten Umfangs dieser Arbeit liegt und daher nicht weiter betrachtet wird (vergleiche 3.3 „Künftigen Erweiterungen vorbehalten").

3.7.2. Evaluation existierender Templatingsprachen

Dieses Kapitel beschäftigt sich mit der Wahl einer Templatingsprache für BlattWerkzeug. Mit dieser soll die `HTML`-Ausgabe durch die Einführung von auswertbaren Ausdrücken, Wiederholungen und bedingten Anweisungen zur Laufzeit beeinflusst werden können. Eine zunächst naheliegende Wahl dafür wären vollwertige Programmiersprachen wie `PHP`, Python oder Ruby. Diese werden jedoch aus den folgenden Gründen nicht berücksichtigt:

- Die Möglichkeit der Definition von eigenen Datenstrukturen, Funktionen, Ausnahmebehandlung, ... sind in `HTML`-Vorlagen nicht von Bedeutung. Analog zu `SQL` müssten bei der Verwendung von vollwertigen Programmiersprachen also wieder eine dem Einsatzzweck angemessene „Teilsprache" definiert werden.

- BlattWerkzeug möchte eine klare Trennung von Präsentations- und Anwendungslogik fördern, die Mächtigkeit dezidierter Programmiersprachen verwischt diese Grenze jedoch: Plötzlich könnten Operationen wie Aggregation nicht nur in SQL ausgedrückt werden, sondern auch durch Schleifen im Template. Und Veränderungen am Datenbestand nehmen nicht mehr zwingend den Weg über eine in BlattWerkzeug definierte mutierende Abfrage, sondern werden direkt in das Template eingebettet.
- Die Mächtigkeit dieser Sprachen geht fast zwangsläufig mit einer inkonsistenten Syntax zur Einbettung von Ausdrücken einher. Zwar kennen viele Skriptsprachen spezielle Modi in denen das Ergebnis eines Ausdrucks direkt in das Dokument geschrieben wird[20], in Einzelfällen, insbesondere bei Iterationen, muss dann allerdings doch mit Funktionen wie echo oder print gearbeitet werden.
- Der große Umfang dieser Sprachen führt zu Einschränkungen, die einem Lernenden willkürlich vorkommen müssen. Zum Beispiel handelt es sich bei dem Bezeichner „class“ in vielen Sprachen um ein reserviertes Schlüsselwort, welches daher an bestimmten Orten einfach nicht verwendet werden darf. Die Details der gewählten Templatingsprache ziehen sich dann entweder bis ins Datenmodell (Schlüsselwörter der Sprache könnten dort schlicht verboten werden) oder erfordern spezielle syntaktische Konstrukte.

Letztendlich liegen diese Sprachen aus gutem Grund außerhalb des abgesteckten Umfangs dieser Arbeit (vergleiche 3.3 „Künftigen Erweiterungen vorbehalten“): Eine vernünftige Integration mit vollwertigen Programmiersprachen, und sei es auch „nur“ für die Templating-Logik, wirft schlichtweg zu viele weitere Fragen auf.

Stattdessen soll an dieser Stelle eine Domänenspezifische Sprache zum Einsatz kommen. Diese dezidierten Templatingsprachen fügen sich so knapp und organisch wie möglich in den umliegenden HTML-Kontext ein und erlauben keine komplizierte Logik. Und auch wenn BlattWerkzeug zu einem späteren Zeitpunkt nicht mehr nur SQL als alleinige Sprache für die Anwendungslogik vorsehen sollte, wird die klare Trennung zwischen Programmlogik und Oberflächendarstellung aller Voraussicht nach beibehalten werden.

Es gibt eine Vielzahl unterschiedlicher Textauszeichnungssprachen für Webseiten, nicht alle eignen sich allerdings, um dynamisch zur Laufzeit Seiten zu erzeugen. Für BlattWerkzeug wurden darüber hinaus zunächst die folgenden Kriterien angelegt, um die

[20]In PHP ist beispielsweise <?= $var ?> in etwa äquivalent zu <?php echo $var ?>.

Anzahl verfügbarer Sprachen schnell zu reduzieren:

Datenanbindung
: An vordefinierten Stellen im Quelltext müssen zur Laufzeit dynamische Werte eingesetzt werden können.

Kontrollstrukturen
: Es muss möglich sein, unterschiedliche Ausgaben in Abhängigkeit von zur Laufzeit ausgewerteten Bedingungen zu erzeugen und Anweisungen wiederholt auszuwerten.

Freie Verfügbarkeit
: Die Verwendung von proprietären oder kostenpflichtigen Abhängigkeiten in Blatt-Werkzeug ist zumindest im Rahmen dieser Thesis ausgeschlossen.

Einbettung von reinem HTML
: Erfahrenen Entwicklern soll es grundsätzlich möglich sein, ihre `HTML`-Kenntnisse in vollem Umfang anzuwenden. Sofern die Templatingsprache also `HTML`-Bestandteile bereinigt, muss es eine Möglichkeit geben, diese Bereinung lokal zu deaktivieren.

Diese Kriterien filtern reine Textauszeichnungssprachen, wie sie in Foren- oder Kommentarsystemen genutzt werden aus. Weder Markdown[21] noch `BB`-Code[22] oder Textile[23] erlauben nativ die programmatische Einbindung von externen Daten. Die innerhalb der Wikipedia genutzte „Wikisyntax“ erlaubt zwar eine einfache Datenanbindung, kennt aber keine Kontrollstrukturen.

Für eine endgültige Entscheidungen sollen Kandidaten anhand der folgenden Kriterien sowohl qualitativ als auch im Rahmen eines einfachen Scoring-Modells verglichen werden. Die quantitative Auswertung findet sich am Ende dieses Kapitels in Tabelle 3, die qualitative Auswertung findet sich im Anschluss an die Kriterien.

Geringe Technische Komplexität, 5 Punkte
: Da BlattWerkzeug-Seiten serverseitig gerendert werden sollen, ist es nicht notwendig auf eine komplexe Templatingsprache mit Funktionen wie einem virtuellen Document-Object-Model (`DOM`) zurückzugreifen. Es gilt die Prämisse: Je komplizierter das interne Verarbeitungsmodell der Sprache ist, desto mehr unverständliche Fehler können auftreten.

[21] `http://daringfireball.net/projects/markdown/` ist die am häufigsten zitierte Spezifikation
[22] Ein Quasi-Standard ohne offizielle Spezifikation, `https://de.wikipedia.org/wiki/BBCode`
[23] Ein Quasi-Standard ohne offizielle Spezifikation, `https://de.wikipedia.org/wiki/Textile`

Geringe Syntaktische Komplexität, 5 Punkte

Die Syntax sollte konsistent sein und optimalerweise nur sehr wenige Syntaxelemente einführen. Je einfacher die Syntax der Sprache ist, desto näher kann die abstrakte Darstellung im Editor den tatsächlich erzeugten Dokumenten sein.

Verbreitung, 3 Punkte

Einige Templatingsprachen haben Einzug in eine Vielzahl unterschiedlicher Softwareprojekte gefunden, andere sind im Hinblick auf spezielle Frameworks zugeschnitten. Je weiter die Templatingsprache verbreitet ist, desto höher die Chance, dass ein Lernender sie möglicherweise auch einmal außerhalb von BlattWerkzeug einsetzen kann.

Einfache Erweiterbarkeit, 3 Punkte

Perspektivisch sollen die Lernenden anhand von BlattWerkzeug auch in die „normale“ Programmierung eingeführt werden können. Vor diesem Hintergrund wäre es von Vorteil, wenn die Templatingsprache eine einfache Programmierschnittstelle bietet, um eigene Funktionen einzubetten.

Sprachübergreifende Implementierungen, 3 Punkte

Existiert lediglich die Referenzimplementierung einer Templatingsprache oder gibt es auch alternative Implementierungen? Dieser Aspekt ist nicht nur ein guter Indikator für die tatsächliche Verbreitung einer Templatingsprache, er könnte perspektivisch im Falle eines Exports von Projekten aus BlattWerkzeug auch praktische Bedeutung haben.

Diesen Kriterien stellen sich die Templatingsprachen Liquid[24], Angular 2[25] und Handlebars[26]. Liquid wurde schon im Rahmen der vergleicharen Arbeiten (2.4 „Software: Jekyll“) kurz vorgestellt und bot sich dementsprechend als Kandidat an. Da für die Implementierung des BlattWerkzeug-Frontend Angular 2 ohnehin zum Einsatz kommt (siehe 4 „Implementierung“), stellt sich die Frage, ob die Nutzung eines weiteren Frameworks tatsächlich notwendig ist. Möglicherweise könnte durch die Vermeidung einer weiteren Abhängigkeit ja auch Aufwand gespart werden? Handlebars wird unter anderem in den Bibliotheken Meteor und Ember.js verwendet und wurde schlicht aufgrund seiner Popularität ausgewählt.

[24] `http://liquidmarkup.org/`
[25] `https://angular.io/docs/ts/latest/guide/template-syntax.html`
[26] `http://handlebarsjs.com/`

Um die Komplexität der Technik zu bewerten, wird zunächst das Ergebnis des Rendervorgangs betrachtet: Sowohl die Referenzimplementierungen von Handlebars als auch Angular 2 generieren JavaScript, welches die sich ergebende `HTML`-Struktur unmittelbar in den `DOM`-Baum des Browsers injiziert. Angular 2 geht noch einen Schritt weiter und gibt die neue Struktur nicht ungefiltert an den Browser weiter, sondern erstellt mit Hilfe eines speziellen Baumes zur Isolation von Veränderungen einen Satz an inkrementell nötigen Änderungen. Das Ergebnis eines Rendervorgangs mit Liquid ist hingegen ein einfacher String mit dem gerenderten Template.

Und natürlich spielt für die Technik auch die Parametrisierung eine Rolle: Bei Handlebars und Liquid ist es in der Referenzimplementierung möglich, zum Rendern einfach eine Funktion mit zwei Parametern aufzurufen: einmal das Template als normale Zeichenkette und dann die Daten als „normale“ Datenstruktur. Angular 2 stellt wesentlich komplexere Anforderungen, dort muss ein spezielles Datenmodell für die Präsentationsschicht bereitgestellt werden.

Um die Syntax zu vergleichen, soll der in Listing 7 angedeutete Datenbestand für einen Blogeintrag gerendert werden. Dieses Beispiel ergänzt die exemplarische `article`-Tabelle um die Spalten `private` und eine Aufzählung von Kategorien.

```
{
  "query" : {
    "article" : {
      "article_id": 4,
      "author": "Alfred E. Neumann",
      "text": "<p>Dies ist ein Text mit HTML-Bestandteilen</p>",
      "private": false,
      "date": -549676800
    },
    "categories" : [
      { "name": "Huch" },
      { "name": "Hoff" }
    ],
  },
  "project" : {
    "name": "Beispielblog"
  }
}
```

Listing 7: JSON-Darstellung eines exemplarischen Datenbestandes

Dieser Beispielartikel soll in den jeweiligen Sprachen wie folgt visualisiert werden:

- Das Datum soll benutzerfreundlich formatiert werden.

- Die Kategorien eines Beitrages sollen mit einer Schleife durchlaufen werden.
- Der gesamte Beitrag soll nicht angezeigt werden, sofern er privat ist.
- Die `HTML`-Bestandteile in Texten sollen erhalten bleiben.

Die **Liquid**-Syntax (Beispiel-Listing 8) wurde grundsätzlich schon im Rahmen von Kapitel 2.4 „Software: Jekyll“ betrachtet. An dieser Stelle sei daher nur noch ergänzt, dass Liquid von Haus aus `HTML` in `{{}}`-Ersetzungen maskiert. Es ist daher nötig, den Artikeltext in drei geschweiften Klammern zu notieren (Zeile 11), diese Syntax deaktiviert diese Sicherheitsfunktion. Die Formatierung der Datumsangabe funktioniert mit einer Syntax wie sie von `UNIX`-Pipes bekannt ist (Zeile 5), Argumente werden als unbenannte Parameter übergeben.

```
<h1>{{ project.name }}</h1>
{% if !query.article.private %}
  <h2>{{ query.article.title }}</h2>
  <div>Autor ist {{ query.article.author }}</div>
  <div>Beitrag vom {{ query.article.date | date: "%d.%b.%Y" }}</div>
  <ul>
    {% for query.categories %}
      <li>{{ name }}</li>
    {% endfor %}
  </ul>
  {{{ query.article.text }}}
{% else %}
  <h2>Dies ist eine private Seite</h2>
{% endif %}
```

Listing 8: Blogartikel mit Liquid

Die Syntax von **Handlebars** ist der von Liquid bemerkenswert ähnlich und basiert ebenfalls auf geschweiften Klammern, welche sich optisch merklich vom normalen HTML-Inhalt abheben. Für Kontrollstrukturen kommt statt dem `%`-Zeichen von Liquid eine Raute zum Einsatz. Listing 9 zeigt, wie das einfache Datenmodell gerendert werden könnte.

Eine praktische syntaktische Eigenart von Handlebars ist die Tatsache, dass auch Schleifen mit einem `else`-Block für leere oder nicht vorhandene Listen ausgestattet werden könnten. In diesem Beispiel bringt diese Alternative allerdings keinen Mehrwert, weil der `<ul>`-Knoten außerhalb der Schleife notiert werden muss.

Um einen Ausdruck unverändert zu übernehmen, also inklusive aller HTML-Sonderzeichen, muss dieser wie in Liquid von drei geschweiften Klammern umgeben werden (Zeile 5).

Statt der Pipe-Notation verwendet Handlebars eine an konventionelle Funktionsaufrufe angelehnte Syntax. Zu Beginn wird der Name genannt, in diesem Beispiel die nicht standardisierte Funktion `formatTime`, danach folgen durch Leerzeichen getrennte Parameter (Zeile 5).

```
1  <h1>{{ project.name }}</h1>
2  {{ #if !query.article.private }}
3    <h2>{{ query.article.title }}</h2>
4    <div>Autor ist {{ query.article.author }}</div>
5    <div>Beitrag vom {{ formatTime query.article.date "%d.%b.%Y" }}</div>
6    <ul>
7      {{ #each query.article.categories }}
8        <li>{{ this }}</li>
9      {{ /each}}
10   </ul>
11   {{{ query.article.text }}}
12 {{ else }}
13   <h2>Dies ist eine private Seite</h2>
14 {{ /if }}
```

Listing 9: Blogartikel mit Handlebars

Angular 2 verzichtet auf nicht-`XML`-artige Erweiterungen und nutzt dafür eigene Knoten und Attribute. Darüber hinaus muss immer die Richtung der Datenbindung angegeben werden, dafür werden Attribute mit runden oder eckigen Klammern eingeschlossen. Im Beispiellisting 10 kommen allerdings ausschließlich eckige Klammern zum Einsatz, da nicht auf Ereignisse reagiert werden muss.

Kontrollstrukturen müssen entweder in speziellen `<template>`-Knoten notiert werden (Zeile 2) oder mit einem `*`-Präfix als Attribut notiert werden (Zeile 11). In diesem Fall ist die Verwendung der `<template>`-Schreibweise für die oberste Unterscheidung notwendig, weil für den Blogbeitrag kein „zusammenfassender“ Elternknoten eingefügt werden soll. In der unteren Unterscheidung kann der optionale Knoten hingegen als Ganzes angezeigt oder entfernt werden, daher ist hier die kompaktere `*`-Notation möglich. Eine Notation für `else`-Blöcke existiert nicht, komplexere Ausdrücke werden daher typischerweise nicht im Template-Code notiert, sondern bereits vorher in den Daten hinterlegt und dann in einem zweiten `ngIf` negiert.

Auch Angular 2 wird `HTML`-Inhalte im Normalfall maskieren. Um diese Beschränkung zu umgehen, können die entsprechenden Daten entweder in der Programmlogik als „sicher“ markiert werden oder direkt an das `innerHtml`-Attribut gebunden werden (Zeile 9).

```
<h1>{{ project.name }}</h1>
<template [ngIf]="!query.article.private">
  <h2>{{ query.article.title }}</h2>
  <div>Autor ist {{ query.article.author }}</div>
  <div>Beitrag vom {{ query.article.date | date: "%d.%b.%Y" }}</div>
  <ul>
    <li *ngFor="let cat in query.categories">{{ cat.name }}</li>
  </ul>
  <div [innerHtml]="query.article.text"></div>
</template>
<h2 *ngIf="query.article.private">
  Dies ist eine private Seite
</h2>
```

Listing 10: Blogartikel mit Angular 2

Für Handlebars und Liquid existieren Implementierungen in unterschiedlichen Programmiersprachen, zumindest für Ruby, JavaScript und PHP scheinen diese Portierungen auch sinnvoll zu funktionieren. Angular 2 hingegen wird zwar für mehrere Sprachen angeboten, in der Praxis kompilieren diese aber alle zu JavaScript.

Jeder der betrachteten Sprachen macht es möglich, die Funktionalität im Rahmen der jeweiligen Syntax zu erweitern. Für Handlebars und Liquid existieren in allen betrachteten Implementierungen einfache Möglichkeiten Funktionen zu hinterlegen, um neue Schlüsselwörter zu registrieren. Angular sieht zur Erweiterung die Programmierung neuer Komponenten vor, welche sich dann als neue Attribute oder Knoten sehr organisch einbetten lassen. Das ist im Endergebnis zwar sehr ansehnlich, die Definition solcher Komponenten erfordert aber vertiefte Kenntnisse der internen Arbeitsweise von Angular 2.

Für alle hier betrachteten Templatingsprachen gilt ebenfalls, dass sie jenseits von Entwicklerkreisen nicht besonders verbreitet sind. Typische Webseiten benötigen natürlich auch keine umfangreiche Logik, um zum Beispiel einem Benutzer zu ermöglichen, einen Kommentar zu schreiben. Im Internet-Alltag werden die Lernenden von diesen Kenntnissen also wohl nicht profitieren.

Sollten Lernende hingegen selbstständig (und jenseits von BlattWerkzeug) Zeit in die Erstellung eigener Webseiten investieren wollen, werden sie dabei häufiger mit Liquid in Berührung kommen. Programme zum generieren von statischen Webseiten verwenden häufig Liquid als Templatingsprache, das gilt insbesondere für das schon vorgestell-

te Jekyll, welches wiederrum (unter anderem) auch auf Github-Pages[27] zum Einsatz kommt. Dieser konkrete Dienst, also Github-Pages, ist interessant, weil er vollkommen kostenfrei genutzt werden kann. Darüber hinaus wird Liquid in den Software-as-a-Service Angeboten von kommerziellen Anbietern wie Salesforce[28] oder Shopify[29] genutzt. Angular und Handlebars werden hingegen faktisch ausschließlich in entwicklerorientierten JavaScript-Bibliotheken verwendet.

Auch wenn für die Lernenden von der konkreten Syntax der Templatingsprache abstrahiert werden soll, wird es immer wieder zu Momenten kommen, in denen diese Abstraktion löchrig ist[30]. Die häufigste Ursache für ein solches Leck werden dabei Fehlermeldungen sein: Sobald diese auftreten, sollte zur Unterstützung der Fehler im generierten Quelltext angezeigt und möglichst hervorgehoben werden. Im Falle von Angular 2 wären die angezeigten Quellen für die Zielgruppe absolut unverständlich.

Letzten Endes stellt sich daher nur die Frage, ob Handlebars oder Liquid zum Einsatz kommen sollte: Angular 2 ist sowohl technisch als auch syntaktisch sehr kompliziert und eignet sich daher nicht besonders gut als „erste Templatingsprache“. Die qualitative Entscheidung fiel dabei zugunsten von Liquid und berücksichtigt vor allem die geringe technische Komplexität. Diese Entscheidung wird von der quantitativen Auswertung in Tabelle 3 in jeder Einzeldisziplin bestätigt. Die Tatsache, dass die Referenzimplementierung von Liquid wie der BlattWerkzeug-Server in Ruby entwickelt wird, ist ein netter zusätzlicher Faktor. Er wird zwar von der quantitativen Auswertung nicht berücksichtigt, gesucht wurde schließlich die „allgemein beste“ Lösung, erleichtert aber erfreulicherweise zusätzlich die Implementierung.

	Liquid	Handlebars	Angular 2
Einfache Technik	5	3	2
Einfache Syntax	5	5	3
Verbreitung	3	3	1
Erweiterbar	3	3	2
Übergreifend	3	2	0
	19	15	8

Tabelle 3: Scoring-Modell für evaluierte Templatingsprachen

[27] Dokumentation zum Hosting verfügbar unter `https://pages.github.com/`

[28] `https://support.desk.com/customer/portal/articles/2916-list-of-liquid-variables`

[29] `https://help.shopify.com/themes/liquid`

[30] `http://www.joelonsoftware.com/articles/LeakyAbstractions.html`

Ebenfalls nicht entscheidend, aber erfreulich, ist der Umstand, dass mit Liquid beliebige Textdateien angereichert werden können, da die Struktur des umgebenden Dokuments vom Liquid-Renderer ignoriert wird. Dieser kümmert sich lediglich um die eingebetteten Anweisungen und behandelt alle Teile dazwischen als einfachen `string`. Für BlattWerkzeug hat dieser Umstand vor allem zur Folge, dass für andere Arten von Dokumenten nicht zwingend eine weitere Templatingsprache notwendig wäre. Zum Beispiel wäre ebenfalls die Bereitstellung einer „entwickler-generierten“ `robots.txt`[31] oder Sitemap[32] möglich. Mit rein auf die Generierung von `HTML` ausgelegten Sprachen wäre dieser Einblick nicht möglich.

3.7.3. Die Templatingsprache Liquid

Auch wenn die Schüler mit der Syntax der verwendeten Templatingsprache in der finalen Version von BlattWerkzeug nur oberflächlich in Berührung kommen sollen, wird Liquid an dieser Stelle sehr knapp und formal vorgestellt. Aufbauend auf dieser Vorstellung wird dann untersucht, wie die entsprechenden Funktionalitäten den Lernenden gezielt vermittelt werden können. Und abgesehen davon sollen Schüler den gesamten Umfang der Sprache bei Bedarf (oder Interesse) perspektivisch auch textuell nutzen können.

Grundsätzlich wird in Liquid zwischen drei verschiedenen Arten von Anweisungen mit jeweils eigener Syntax unterschieden:

Objekte

In das Dokument eingebettete Bezeichner werden in doppelt geschweiften Klammern notiert (`{{Objekt}}`) und dann zur Laufzeit durch den hinterlegten Wert ersetzt. Konstante Strings werden in Anführungszeichen notiert, Zahlen und boolesche Werte (also `true` und `false`) stehen für sich selbst.

Tags

Tags steuern den Programmfluss mit Anweisungen wie `if` und `for`. Um sie von interpolierten Werten zu unterscheiden, werden diese Anweisungen in `{% %}`-Klammerpaaren notiert.

[31] Diese Datei wird von Suchmaschinen ausgewertet, um gewisse Teile der Seite von der Indexierung auszuschließen.

[32] Typischerweise ein XML-Dokument, welches alle zu indexierenden Seiten mit einigen Metadaten, zum Beispiel der Frequenz von Veränderungen, auflistet.

Filter

Mit Filtern können in Ausdrücken genutzte Daten manipuliert werden. Sie funktionieren nach dem gleichen Prinzip wie Pipes auf der Kommandozeile und ermöglichen es, die Repräsentation von Daten zu verändern. Jeder Wert kann über einen Filter modifiziert werden, der Ausdruck `{{ 'miXeD cAse' | downcase }}` würde also zu `'mixed case'` ausgewertet werden.

Alle diese Anweisungen beziehen sich immer auf einen vordefinierten Datenbestand, welcher dem Renderer zusammen mit dem eigentlichen Template übergeben wird. Dieser Datenbestand gliedert sich in Name-Wert-Paare, Listen und atomare Werte, die Navigation erfolgt mittels des `.`-Operators für Zugriffe auf benannte Schlüssel oder mit dem `[]`-Operator für einzelne Werte eines Arrays.

Verschiedene Werte können mit den typischen Vergleichsoperatoren (`==`, `!=`, `>`, `<`, `>=`, `<=`) zu booleschen Ausdrücken kombiniert werden. Für Zeichenketten existiert noch der infix-notierte `contains`-Operator, um zu prüfen, ob eine Zeichenkette in einer anderen enthalten ist. Mehrere Ausdrücke können dann ihrerseits mit `and` und `or` kombiniert werden. Dabei werden stets die mit `and` verknüpften Ausdrücke vor jenen, die mit einem `or` verknüpft sind, ausgewertet. Der Ausdruck (`true or false and false`) wird folglich nicht von links nach rechts ausgewertet, sondern ergibt insgesamt `true`. Diese Reihenfolge kann nicht beeinflusst werden, Klammern in Liquid-Ausdrücken sind ein Syntaxfehler.

Neben dem schon häufig erwähnten `if`-Tag stehen zur Beeinflussung des Kontrollflusses aber noch einige weitere Tags bereit: Verkettungen sind über `elsif`- und `else`-Tags möglich, bei `unless` handelt es sich schlichtweg um ein negiertes `if`. Und wie in anderen Programmiersprachen existiert auch in Liquid ein `case`- / `when`-Konstrukt, um auf verschiedene Ausprägungen eines einzelnen Wertes zu reagieren.

Iteriert wird in Liquid hauptsächlich mit dem schon oberflächlich betrachteten `for`-Tag. Die grundsätzliche Syntax lautet dabei „`for` *`varname`* `in` *`collection`*“, wobei `varname` der Bezeichner der Iterationsvariable ist und `collection` der Container, über den iteriert werden soll. Auf ein `for`-Tag kann ein `else`-Tag folgen, welches dann im Fall eines leeren Containers angezeigt wird.

Anfang und Ende der Iteration kann über die Eigenschaften `offset` und `limit` gesteuert werden. Mit der Angabe „`for item in collection offset:20, limit:10`“ kann zum

Beispiel von Element #20 an über maximal 10 weitere Elemente iteriert werden. Mittels der Angabe `reversed` kann die Iterationsreihenfolge umgedreht werden.

In Schleifen ist ein Objekt names `forloop` verfügbar. Dieser verfügt in Abhängigkeit vom aktuellen Stand der Iteration über Eigenschaften wie `first`, `last` und `index`, um während der Iteration auf „spezielle Durchläufe“ reagieren zu können.

Das wesentliche Tag zur Strukturierung der eigenen Webseite in logische Komponenten ist die `include`-Anweisung. Diese funktioniert ähnlich wie ein Funktionsaufruf mit Paremetern. Die konkrete Syntax für einen Aufruf wird in Listing 11 demonstriert, als aufgerufenes Template wird eine Liste von Kategorien gerendert (Listing 12). Mit dieser Anweisung können also in unterschiedlichen Kontexten identisch zu rendernde Bestandteile ausgelagert werden. Für den „Beispielblog“ müsste die Aufzählung von Kategorien nicht mehr für Übersichts- und Detailseite dupliziert werden.

```
{% include 'categories' categories: query.article_categories %}
```

Listing 11: Verwendung der Liquid-Anweisung `include`

```
{% if categories.length > 0 %}
  <ul>
    {% for category in categories %}
      <li>{{ category.name }}
    {% endfor %}
  </ul>
{% else %}
  Keine Kategorien
{% endif %}
```

Listing 12: `categories.liquid`, Template für die Liquid-Anweisung `include`

Perspektivisch wäre es natürlich interessant diese Mechanik auch unmittelbar in Blatt-Werkzeug verfügbar zu machen. Für den Augenblick ist diese Anweisung aber vor allem für die Implementierung relevant. Die in Kapitel 3.7.6 „Komplexe Bedienelemente“ beschriebenen Funktionen könnten als Liquid-Templates bereitgestellt werden. Darüber hinaus ließen sich mit diesem Ansatz auch „übergeordnete“ Seiten erstellen, wie sie oftmals für ein einheitliches Layout notwendig sind.

Bei den Filtern liegt eine funktionale Überschneidung mit den Projektionen in `SQL` vor, insbesondere im Hinblick auf die Textfunktionen (siehe 9e in Kapitel 3.5.2 „Mögliche Einschränkungen der Ausdrücke“). Alle dort beschriebenen Funktionen könnten auch

mit Liquid ausgedrückt werden. Und darüber hinaus kennt Liquid noch ein paar weitere Filter wie zum Beispiel `capitalize` (wandelt jeden Wortanfang in Großbuchstaben um), die keine Entsprechung in SQLite haben. Im Sinne einer Trennung von Datenmodell und Anzeigelogik ist es im Regelfall sinnvoll, diese Arten von Transformationen mit Liquid vorzunehmen, letzten Endes ist das aber eine Entscheidung der Lehrperson.

Die Einbindung von den schon erwähnten Textauszeichnungssprachen, wie zum Beispiel Markdown, könnte ebenfalls über Filter erfolgen. Für das Blogbeispiel wäre es folglich ein leichtes, zwar „echtes“ `HTML` in den Artikeln zuzulassen, in den Kommentaren hingegen nur eine von Textauszeichnungssprachen zugelassene `HTML`-Teilmenge ohne gefährliche Inhalte zu erlauben.

3.7.4. Layout

Eine immer wiederkehrende Frage bei der Entwicklung von Webseiten ist die nach dem Layout der gesamten Seite. BlattWerkzeug orientiert sich explizit an dem „Responsive Design“-Paradigma, anstatt die Seiten für eine bestimmte Bildschirmgröße zu optimieren. Letztendlich soll sich eine gerenderte `HTML`-Seite auf mobilen Geräten also ebenso „richtig“ anfühlen wie auf „klassischen“ Computern.

Technisch gesehen erfolgt die Umsetzung eines responsiven Layouts über eine Vielzahl unterschiedlicher `CSS`-Direktiven, deren Erstellung oder Bearbeitung keinesfalls ein Kernbereich von BlattWerkzeug ist. Allerdings lassen sich auf den meisten Seiten immer wiederkehrende Strukturen feststellen: die Unterteilung der Seite in ein Raster aus Zeilen und Spalten. Diese Layoutstruktur wird mittlerweile von einer Vielzahl von vorhandenen `CSS`-Bibliotheken und den `HTML5`-Flexboxen unterstützt und soll auch das grundsätzliche Gedankenmodell für die hier beschriebenen Oberflächen sein. Es ergeben sich daraus die folgenden Layoutelemente:

Zeilen
: Zeilen enthalten eine oder mehrere Spalten, wobei jeder Inhalt zwingend in einer Spalte platziert werden muss. Die Höhe einer Zeile entspricht der Höhe der höchsten Spalte.

Spalten
: Die Breite einer Spalte wird in einem relativen Maß mit Bezug zu einer bestimmten Bildschirmgröße angegeben. So können für breite Bildschirme z.B. drei Spalten

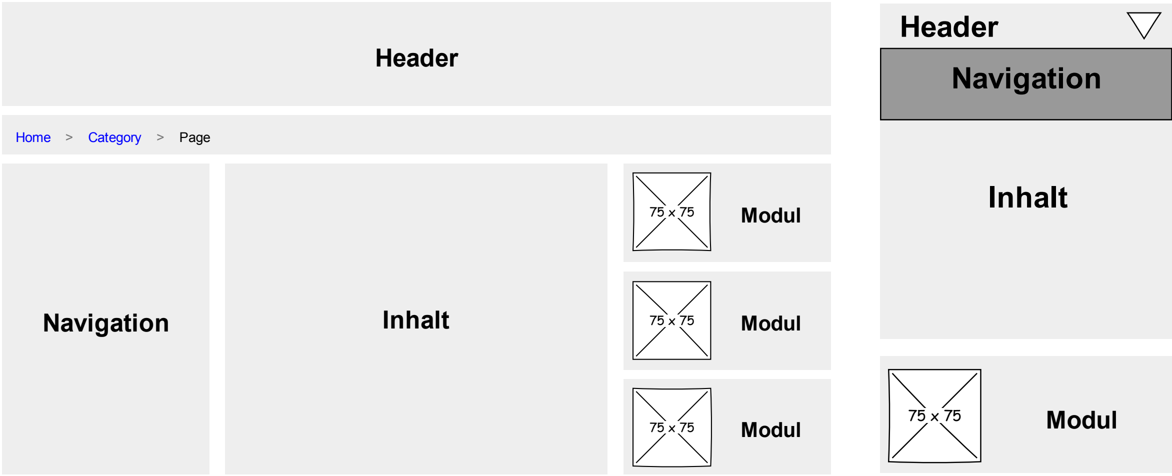


Abbildung 23: Identische Layoutstruktur auf großen und kleinen Bildschirmen. Der Navigationsbereich wird für den kleinen Bildschirm nur auf Aufforderung eingeblendet.

> vorgesehen werden, für sehr schmale Geräte wie Smartphones hingegen nur eine Spalte. Sollte die Summe dieser Breitenangaben der Spalten die maximale Breite einer Zeile überschreiten, werden diese auf mehrere Zeilen umgebrochen.

Abbildung 23 zeigt das identische Layout aus Zeilen und Spalten in der Darstellung auf unterschiedlich großen Anzeigegeräten. Auf keinem der Geräte werden die beiden obersten Zeilen für den Header und die Pfadangabe weiter in Spalten unterteilt. Diese bestehen technisch gesehen also aus einer einzigen, sehr breiten Spalte.

Die dritte Zeile enthält drei Spalten (Navigation, Inhalt und eine Leiste für Module), wobei diese letzte Modulspalte wiederum mehrere Zeilen enthält. In der mobilen Ansicht wird die Navigationsleiste möglicherweise dynamisch verborgen und kann mit dem Dreieck im Header angezeigt werden. Die Module wandern in dieser Ansicht unter den Inhalt.

3.7.5. `HTML`-Bedienelemente

Grundsätzlich lassen sich alle Bedienelemente anhand der Struktur von Ein- und Ausgaben unterscheiden (Abbildung 24). Die Unterscheidung, ob eine Ein- bzw. Ausgabe exakt einen Wert oder alternativ mehrere (oder keinen) Wert(e) erwartet bzw. liefert, ist für die Oberfläche von zentraler Bedeutung. Die Kapitel 3.7.7 „Bindung von Abfragen

	Keine Eingabe	Eingabe
Eine Zeile	Unmittelbare Ausgabe, z.B. in einem Text	Einfaches Eingabeelement, z.B. eine Textbox
Beliebig	Wiederholte Ausgabe, z.B. eine Tabelle	Eingabeelement mit Mehrfachauswahl, z.B. eine Liste von Checkboxen

Abbildung 24: Einordnung von Bedienelementen

an die Oberfläche“ und 3.7.8 „Eingabe von Daten über die Oberfläche“ erläutern diese Unterscheidung noch einmal ausführlich.

Zwar sollen im Regelfall keine komplett statischen Seiten angezeigt werden, trotzdem muss es natürlich eine Möglichkeit geben statische Texte ohne Bindung an irgendwelche Abfragen zur Anzeige zu bringen. Die hier vorgestellten Bedienelemente funktionieren auch ohne jede Datenquelle, erlauben aber auch die Einbettung von Liquid-Objekten in den Text.

Überschriften
: `HTML` sieht die Verwendung von Überschriften in sechs Hierarchiebenen vor und stellt dafür distinkte Elemente zur Verfügung (`<h1>` bis `<h6>`). Dieser Umstand soll nicht unmittelbar abgebildet werden. Stattdessen gibt es ein allgemeines Bedienelement “Überschrift”, zu dem sich dann eine Ebene angeben lässt.

Absätze
: Blöcke von zusammenhängendem Text werden als Absatz ausgezeichnet, das entsprechende `HTML`-Äquivalent ist das `<p>`-Element.

Links
: Um dem Endbenutzer die Navigation von einer Seite zur nächsten zu ermöglichen, kommen üblicherweise Links in Form von `<a>`-Elementen zum Einsatz. Wenn das

Ziel des Links, also das `href`-Attribut, eine Seite des gleichen Projektes ist, sollte BlattWerkzeug die Gültigkeit der Angaben validieren.

Formulare, Eingaben & Knöpfe

Immer wenn vom Benutzer Werte abgefragt werden sollen, kommen dabei Formulare zum Einsatz. Auf diesen können mittels Eingabefeldern (`<input>`) und Auswahlelementen (`<select>`) Benutzereingaben getätigt werden. Aktionen werden dann mit einem Klick auf einen Knopf (`<button>`) ausgelöst, was wiederrum eine serverseitige Operation oder eine Navigation auslösen könnte.

Bilder

Zwar syntaktisch betrachtet ein eher simples Unterfangen, für Schüler erfahrungsgemäß aber motivierend, ist die Einbindung von Bildern. Diese entsprechen einem `HTML`-`<img>`-Element, für welches das `src`-Attribut gesetzt werden muss.

3.7.6. Komplexe Bedienelemente

Viele im Desktopbereich „typische“ Bedienelemente haben in reinem `HTML` keine Entsprechung. So gibt es zum Beispiel jenseits der `alert()` und `input()` Funktionen keine standarisierte Möglichkeit, modale Dialoge anzuzeigen. Stattdessen existieren je nach Einsatzzweck eine Vielzahl von Bibliotheken. So genannte „Lightboxen“ kommen zum Einsatz, um Bilder nach einem Klick großformatig anzuzeigen, ohne dafür die aktuelle Seite zu verlassen. Und so ziemlich jedes größere `HTML`-Framework bietet Möglichkeiten, faktisch beliebige `HTML`-Strukturen im Vordergrund anzuzeigen.

Trotzdem haben sich viele Endanwender an komplexe Webanwendungen gewöhnt, deren Umsetzung um die Jahrtausendwende im Browser vollkommen undenkbar gewesen wäre. Diese Erwartungshaltung bringt BlattWerkzeug in ein Dilemma: Gibt man dieser Erwartungshaltung nach und bietet zahlreiche „künstliche“ Bedienelemente für Spezialfunktionen an, hat das mit der Vermittlung von `HTML`-Kenntnissen nur noch am Rande etwas zu tun. Andererseits sollen die Lernenden mit „praktisch vorzeigbaren Ergebnissen“ motiviert werden.

Um solche Arten von Bedienelementen umzusetzen, werden komplexe `CSS`-Eigenschaften für `HTML`-Block- oder -Inline-Elemente (also `<div>` und `<span>`) verwendet. Diese beiden Elemente haben keine unmittelbare inhaltliche Bedeutung, sondern ziehen diese aus den gesetzten Klassen oder IDs (also den Attributen `class` und `id`). Im Rahmen dieser

Thesis wird die Unterstützung von diesen CSS-orientierten Elementen aber nicht weiter betrachtet (siehe 3.3 „Künftigen Erweiterungen vorbehalten“).

Auch die semantisch bedeutungsvollen HTML-Elemente können aber durchaus kompliziert ausfallen. Dieses Spannungsfeld lässt sich gut anhand eines für BlattWerkzeug zentralen Bedienelements diskutieren: der Tabelle. Aus didaktischer Sicht scheint es sinnvoll, den Lernenden dieses Element möglichst frühzeitig vorzustellen. Immerhin wird bei den Abfragen mit tabellarischen Daten gearbeitet, da liegt es nahe, diese Daten in der Oberfläche zunächst auch auf genau diese Art und Weise zu präsentieren.

Allerdings handelt es sich bei Tabellen um eine der deutlich komplizierteren HTML-Konstrukte. Es besteht typischerweise aus zwei unmittelbaren Kindelementen, nämlich `<thead>` für die Spaltenüberschriften und `<tbody>` für die eigentlichen Daten. Für neue Zeilen kommen dann die „Table Row“-Elemente `<tr>` zum Einsatz, Zellen werden in den Überschriften dann in `<th>`-Knoten („Table Head“) und im Rumpf mit `<td>`-Knoten („Table Data“) notiert. Und natürlich muss die Anzahl dieser durch Zellenknoten definierten Spalten über alle `<tr>`-Elemente hinweg identisch sein[33]. Erfahrungsgemäß bereitet es Anfänger auch immer wieder Probleme, dass die in der Tabelle nebeneinander angezeigten Zellen im HTML-Quelltext typischerweise untereinander notiert werden. Und damit nicht genug: Um wirklich alle Zeilen einer Abfrage anzuzeigen, ist die Verwendung von geschachtelten Schleifen für Zeilen und Spalten notwendig. Listing 13 zeigt exemplarisch, wie der nötige Quelltext aussehen könnte.

Mit Liquid bietet sich allerdings auch die Möglichkeit, diesen Code weitestgehend vor den Lernenden zu verstecken. Listing 14 illustriert, wie mit der Verwendung des `include`-Tags die Einbindung der Tabelle auf die Angabe der darzustellenden Zeilen und Spalten reduziert wird. Dabei handelt es sich natürlich nicht um die exakte Darstellung, wie sie innerhalb eines Drag & Drop-Editors dargestellt werden sollte.

3.7.7. Bindung von Abfragen an die Oberfläche

Die vergangenen Kapitel haben demonstriert, wie mit Liquid auf prinzipiell beliebige Datenstrukturen zugegriffen werden kann. Aber wie sollten die Ergebnisse der verfügbaren Abfragen bereitgestellt werden? Auf abstrakter Ebene ist die Beantwortung dieser

[33] Eine mögliche Ausnahme hiervon wäre theoretisch das `colspan`-Attribut. Dadurch wird die Implementierung jedoch lediglich noch komplizierter.

```
<table>
  <thead>
    <tr>
      {% for name in data.columnNames %}
        <th>{{ name }}</th>
      {% endfor %}
    </tr>
  </thead>
  <tbody>
    {% for row in data.rows %}
      <tr>
        {% for name in data.columnNames %}
          <td>{{ row[name] }}</td>
        {% endfor %}
      </tr>
    {% endfor %}
  </tbody>
</table>
```

Listing 13: Code für eine `HTML`-Tabelle mit Datenanbindung

```
{% include "query_table"
     data: query.alle_kommentare,
     columns: ["spalte_a","spalte_b"] %}
```

Listing 14: Code für eine Liquid-Tabelle mit Datenanbindung

Frage eindeutig: Da das Ergebnis einer `SQL`-Abfrage grundsätzlich eine Tabellenstruktur mit Zeilen, Spalten und Zellen ist, wird dieses Datenmodell auch für die Oberfläche zugrunde gelegt. Die an die Bedienelemente zu bindenden Daten sind also grundsätzlich tabellarischer Natur. Dabei bedürfen einige Details bezüglich der Benennung und der Indizierung jedoch besonderer Beachtung.

Um im Falle von mehreren in Frage kommenden Abfragen eine eindeutige Zuordnung vornehmen zu können, müssen Abfragen, deren Ergebnis an die Oberfläche gebunden werden soll, mit einem eindeutigen Namen bezeichnet werden. Dieser Name kann von den Entwicklern frei gewählt werden, sollte aber analog zu einem Funktionsbezeichner sinnvoll darüber Auskunft geben, was eine Abfrage bewirkt. Diese Trennung zwischen dem Namen der Abfrage innerhalb des Projekts und dem Namen innerhalb des `query`-Namensraums erlaubt die mehrfache Verwendung von ein und derselben (vermutlich unterschiedlich parametrisierten) Abfrage auf der gleichen Seite.

Der Umgang mit den Zeilen erfordert keine Sonderbehandlung, hier genügt die Angabe eines Zeilenindexes. Im Normalfall wird die Iteration sowieso nicht explizit über den

Index, sondern implizit über die Elemente stattfinden. Für die Spalten entfällt glücklicherweise eine „künstliche“ Benennung, diese werden trivialerweise über ihren Namen entsprechend der `SELECT`-Komponente angesprochen. Mit diesen drei Werten (Name der Abfrage, Zeilenindex, Spaltenname) lässt sich jede Zelle einer Menge von Abfragen eindeutig identifizieren.

Aus praktischen Gründen sollten Abfragen allerdings bezüglich der Anzahl der erwarteten Zeilen annotiert werden, konkret unterschieden werden muss dabei zwischen „exakt eine Zeile“ und „beliebig viele (auch keine) Zeilen“. Im Falle einer Abfrage mit nur einer Zeile entfällt dann die Notwendigkeit einen Zeilenindex anzugeben, dieser bezieht sich dann implizit immer auf die einzige, zur Verfügung stehende Zeile. Wenn Daten an die Oberflächen gebunden werden, entfällt durch diese Abkürzung die permanente Angabe einer redundanten Information. Listing 15 zeigt eine Abfrage, die in Listing 16 und 17 einmal mit und einmal ohne Index gebunden wird. Die Schreibweise ohne Index kommt der Lesbarkeit eindeutig zu gute.

```
SELECT caption, text
FROM   article
LIMIT  1;
```

Listing 15: Abfrage mit maximal einer Ergebniszeile

```
<!-- query.article = Listing 15: Abfrage mit maximal einer Ergebniszeile -->
{{ query.article[0].name }}, beginnt mit den folgenden Worten {{ query.
   article[0].text | truncatewords: 4 }}.
```

Listing 16: String-Interpolation mit Indexzugriff

```
<!-- query.article = Listing 15: Abfrage mit maximal einer Ergebniszeile -->
{{ query.article.name }}, beginnt mit den folgenden Worten {{ query.
   article.text | truncatewords: 4 }}.
```

Listing 17: String-Interpolation mit implizitem Index

Die Annotation „Abfrage betrifft genau eine Zeile“ muss jedoch vom Entwickler manuell im `SQL`-Editor gesetzt werden, eine automatische Berechnung wäre für beliebige Abfragen unmöglich[34] und auch für speziell strukturierte Abfragen ist diese Unterscheidung zumindest nicht trivial. Oder um es anders auszudrücken: Die Erweiterung der Entwicklungsumgebung um diese automatische Vorhersage wäre Stoff für eine weitere wissenschaftliche Arbeit, keinesfalls aber Gegenstand dieser Arbeit. Praktischerweise lässt sich

[34] Viele `SQL`-Dialekte sind Turing-Vollständig und schon das Halteproblem ist nicht entscheidbar.

eine auftretende Verletzung dieser Annotation zur Laufzeit aber sehr gut erkennen und daher auch unmittelbar an den Entwickler oder Endanwender kommunizieren.

Abfragen mit beliebig vielen Ergebniszeilen bedürfen zur Darstellung einer Schleife oder eines dezidierten komplexen Bedienelements. Auch hier könnte im Einzelfall die Behandlung eines leeren Ergebnisses sinnvoll sein, dann aber „nur“ aus ästhetischen Gründen: Eine Schleife, die nicht ein einziges Mal durchlaufen wird, ist auch in Liquid kein Fehler.

Diese gesonderte Behandlung von bestimmten Tabellendimensionen ist übrigens keine Erfindung dieser Thesis: Auch `SQL` selbst hebt einige Abfragen mit speziellen Strukturen hervor. Tabellen mit nur einer Zelle können als skalarer Wert eingesetzt werden, bei nur einer Spalte können Abfragen z.B. für `IN`-Ausdrücke verwendet werden.

3.7.8. Eingabe von Daten über die Oberfläche

Immer, wenn eine Abfrage mit Platzhaltern an eine Seite gebunden wird ist es notwendig, dem Benutzer eine Möglichkeit zu geben, diese Platzhalter zu füllen. Benutzereingaben sind in `HTML` nur zulässig, wenn sie innerhalb eines `<form>`-Elements platziert werden. Für diese Formulare müssen entweder global im `<form>`-Element oder als Teil der vom Benutzer gedrückten Knöpfe stets die Methode der Übertragung (`method`, eines der `HTTP`-Verben `GET` oder `POST`) und eine Ziel-`URL` (`action`) angegeben werden. Diese Optionen zur Navigation werden im Rahmen von Kapitel 3.7.9 „Navigation“ besprochen.

Jedes Kindelement eines Formulars muss mit einem Namen ausgestattet werden. Aus diesem Namen und dem vom Benutzer eingegebenen oder ausgewähltem Wert wird dann ein Schlüssel-Wert-Paar gebildet, welches zusammen mit der jeweiligen Anfrage versendet wird.

Im einfachsten Fall kann diese Bindung über simple `<input type="text">`-Elemente hergestellt werden. Das ist für viele Arten von Such- und Eingabemasken auch sicherlich ausreichend, führt jedoch zu Problemen bei Beziehungen zwischen Entitäten. In diesem Fall müsste in das Textfeld vom Benutzer ein Primärschlüssel eingegeben werden, was ganz sicher kein besonders benutzerfreundliches Konzept darstellt.

```
<!-- query.alle_artikel = "SELECT * FROM article" -->
<select>
  {% for article in query.alle_artikel %}
    <option value="{{ article.article_id }}">
      {{ article.article_id }}
    </option>
  {% endfor %}
</select>
```

Listing 18: Containerelemente mit Kindern

In einem solchen Fall soll es also möglich sein auch Bedienelemente wie Comboboxen zu verwenden, die ihrerseits wieder eine Abfrage zur Anzeige der zur Verfügung stehenden Daten benötigen. Dabei muss klar zwischen zwei benötigten Informationen unterschieden werden. Auf der einen Seite muss das Resultat eines solchen Bedienelements einen identifizierenden Wert zurückliefern, aller Regel nach vermutlich einen Primärschlüssel. Auf der anderen Seite erwarten die Benutzer aber die Anzeige von „intuitiven" Informationen, anhand derer sie ihre Auswahl treffen können. Der Primärschlüssel ist hingegen üblicherweise eine künstliche Spalte, welche für den Endanwender keinerlei Informationsgehalt hat.

Praktischerweise ist auch diese Unterscheidung in `HTML` vorgesehen: Bei einem `<select>`-Element wird explizit zwischen dem internen Wert (das Attribut `value`) und der Repräsentation für den Benutzer (der Inhalt des Elements) unterschieden. Listing 18 demonstriert, wie diese Informationen mit Liquid im `<option>`-Element als Kinder eines `<select>`-Elements beschrieben werden. Die Darstellung in Browsern erfolgt in der Regel über eine Combobox.

3.7.9. Navigation

Die Navigation auf Webseiten teilt sich in zwei unterschiedliche Problemstellungen: Noch vor dem Rendervorgang stellt sich die Frage, zu welcher konkreten Seite eine beliebige `URL` aufgelöst werden soll. Und während der Generierung einer Seite müssen sich Verweise auf andere Seiten erzeugen lassen können.

Damit für jede relevante URL eindeutig ist, zu welcher Seite sie aufgelöst werden soll, könnte man in BlattWerkzeug eine zentrale Routing-Konfiguration hinterlegen. In dieser wird einer Menge von Pfaden, welche möglicherweise auch Platzhalter für Parameter vorsehen, mit den entsprechenden Zielseiten verknüpft. Solche Pfadangaben könnten für

das Blogbeispiel wie in Listing 19 angedeutet repräsentiert werden. Bei dem `selector` handelt es sich um ein Prädikat, welches im Falle einer positiven Auswertung darauf hinweist, dass die entsprechende `page` dargestellt werden soll.

```
[
  { "selector": "/",             "page": "Hauptseite" },
  { "selector": "/beitrag/:id",  "page": "Artikel" },
]
```

Listing 19: Zentrales Routing für das Blog-Beispiel

Dabei bedürfen zwei Details besonderer Beachtung: Parameter in der `URL` werden mit der `:parametername`-Notation eingeführt[35]. Außerdem handelt es sich um eine Liste von Regeln, die Reihenfolge der Einträge ist folglich von Bedeutung. Wenn ein hypothetischer Eintrag mit dem Selektor `/beitrag/test` eingeführt werden sollte, muss das auf jeden Fall vor dem Selektor `/beitrag/:id` geschehen. Andererseits passt der Parameter `:id` auf das `URL`-Segment `test` und maskiert diese Route. Unter dieser Randbedingung lässt sich dann aber die Abbildung `URL` → `Seite` eindeutig berechnen.

In diesem Datenmodell können allerdings Inkonsistenzen entstehen, wenn sich die Anzahl oder Benennung der für eine Seite erforderlichen Parameter verändert. Die Behandlung solcher Fehler erfordert allerdings grundsätzlich die Unterstützung des Entwicklers: Nur er kann entscheiden, ob eine umbenannte Variable auch unter neuem Namen semantisch gleichwertig ist. Es handelt sich also um ein Problem, dem am ehesten durch eine passende Benutzeroberfläche beizukommen ist.

Nachdem die zu rendernde Seite bekannt ist, stellt sich folglich die inverse Frage: Wie kann die `URL` zu einer bestimmten Seite generiert werden? Dabei stellt man zunächst fest, dass der `HTML`-Standard zwei unterschiedliche Mechanismen zur Navigation vorsieht. Die naheliegendste Variante ist die direkte Generierung eines `<a>`-Knotens mit dem passenden `href`-Attribut. Mit der Liquid-Syntax lassen sich beliebige Strings generieren, also auch gültige URLs. Allerdings funktioniert dieser Ansatz nur mit Daten, die auf dem Server während des Rendervorgangs bekannt sind. Benutzereingaben in Formularelementen wie `<input>` können in diesem Kontext nicht verwendet werden.

Es ist folglich schon aufgrund der Vorgaben des `HTML`-Standards sinnvoll, die Unterstützung von zwei separaten Mechanismen zur Navigation zu unterstützen: Sofern der

[35] Es handelt sich dabei um die gleiche Notation, wie sie von den Bibliotheken Sinatra oder Angular 2 verwendet wird.

Endanwender eigene Eingaben vornehmen soll, kommt ein `<form>`-Element zum Einsatz. Durch die Nutzung von versteckten Eingabelementen (`<input type="hidden">` können aber auch serverseitig vorhandene Daten, zum Beispiel aus Abfragen oder GET-Parametern innerhalb von Formularen, verwendet werden.

Um den Zugriff auf mutierende Abfragen zu ermöglichen, existiert für jede Seite noch ein spezieller `query/:queryName`-Unterpfad. Wenn eine unter `/beitrag/:artikel_id` erreichbare Seite die Abfrage zum Hinzufügen eines neuen Kommentars (Listing 20) unter dem Namen `add_comment` verfügbar machen möchte, sähe die vollständige Routendefinition aus wie folgt: `/beitrag/:artikel_id/query/add_comment`. Die Parameter der Abfrage (`article_id`, `name` und `text`, das aktuelle Datum wird automatisch berechnet) müssen von der Seite (oder genauer: von dem Formular) bereitgestellt werden.

```
INSERT INTO comment (article_id, commenter_name, text, date)
VALUES (:article_id, :name, :text, strftime('%s','now'))
```

Listing 20: Abfrage zum Hinzufügen eines neuen Kommentars

Abfragen werden immer mittels der `POST`-Methode ausgeführt und haben keine eigene Darstellung. Stattdessen leitet eine ausgeführte Abfrage normalerweise zur aufrufenden Seite zurück, auch eine Umleitung zu einer anderen Seite wäre aber denkbar. Diese Umleitung und die Übertragung via `POST` stellt sicher, dass ein Endanwender durch einfaches Neuladen der Seite eine Abfrage mehrfach ausführt.

3.8. Drag & Drop-Editor für Oberflächen

Nachdem das vorige Kapitel den technischen Unterbau für die mit BlattWerkzeug erstellten Seiten beschrieben hat, stellt sich die Frage, wie man diesen in eine angemessene Oberfläche „verpacken" kann. Diese Suche nach dem richtigen Editor für einen sowohl lehrreichen, als auch produktiven Seiten-Editor, wird sich wohl aller Voraussicht nach nicht mit einer definitiven Antwort abschließen lassen. Die Anzahl an möglichen Herangehensweisen ist immens, das zeigt schon die Vielfalt an existierenden und historischen Produkten zur Erstellung von Webseiten oder auch „normalen" Programmoberflächen. Die wesentlichen Leitfragen für dieses Kapitel sind daher die Folgenden:

1. **Wie funktionieren andere, verbreitete Oberflächeneditoren und was kann BlattWerkzeug von ihnen lernen?** Insbesonders kommerzielle Produkte für

„normale“ Anwender werden im Regelfall auf einen hohen Produktivitätsgrad ausgerichtet sein, nicht auf einen nachhaltigen Lerneffekt. Und trotzdem stellt sich die Frage, nach welchen Mustern diese Werkzeuge funktionieren.

2. **Welches Abstraktionsniveau passt zur Zielgruppe von BlattWerkzeug?** Es gilt eine geeignete Balance zwischen vorzeigbaren Ergebnissen und nachhaltigem Lerneffekt zu finden. Diese Fragestellung hängt eng mit der gewählten Darstellung im Editor zusammen, daher werden die beiden wesentlichen Optionen zur Visualisierung in den kommenden Kapiteln besprochen.

3.8.1. Vor- und Nachteile von WYSIWYG-Darstellungen

Eine typische Methode, um die Einstiegshürde für die Gestaltung von Oberflächen zu senken, ist die Verwendung von WYSIWYG-Editoren. Programmiersprachen wie Delphi oder Visual Basic verdanken einen Großteil ihrer initialen Popularität auch ihren Entwicklungsumgebungen, in denen eben diese Funktionalität prominent unterstützt wird. Diese `IDE`s ermöglichten schon zur Entwicklungszeit eine sehr gute Vorschau auf das endgültige Aussehen der aktuell bearbeiteten Oberfläche. Der zeitaufwändige Kreislauf aus „Wert im Quelltext verändern → kompilieren → ausführen → zu korrekter Maske navigieren → Darstellung überprüfen → Wert erneut anpassen“ wird drastisch verkürzt.

Dieser Ansatz hat aber auch Grenzen: Ohne Kenntnis der genauen Datenbestände bleibt die Vorschau oftmals unvollständig. Und insbesondere bei sehr dynamischen Oberflächen mit vielen optionalen Bestandteilen, welche dann zur Laufzeit ein- oder ausgeblendet werden, im Editor aber stets sichtbar sind, gleicht die Vorschau innerhalb des WYSIWYG-Editors der Endbenutzeransicht dann doch nur noch sehr beschränkt.

Bei der Entwicklung von Desktopanwendungen ist auch heute noch die Verwendung eines WYSIWYG-Editors sehr verbreitet, allerdings werden die Beschreibungen der Oberflächen häufig in spezielle Textdateien ausgelagert. Diese lassen sich dann sowohl mit einem normalen Texteditor, als auch mit spezialisierten Editoren bearbeiten. Moderne Bibliotheken für Benutzerschnittstellen wie JavaFX, Android oder die *Windows Presentation Foundation* (WPF) erlauben mit diesem Ansatz die größtmögliche Flexibilät: Jeder Programmierer kann entscheiden, ob er mit der Beschreibungssprache, dem grafischen Werkzeug oder beiden Herangehensweisen arbeiten möchte.

Auch zur Pflege und Generierung von Webseiten existieren und existierten eine Vielzahl unterschiedlicher WYSIWYG-Editoren wie Microsoft Frontpage, Adobe Dreamweaver, ... Diese spielen als Lösungen für hobbymäßige Webseitenbetreiber mittlerweile eine eher untergeordnete Rolle. Heute gängige Lösungen für den Betrieb eigener Webseiten bieten dem Endanwender als primäre Schnittstelle eine im Web verfügbare, administrative Oberfläche. Die meisten quelloffenen Content-Management-Systeme wie Wordpress, Joomla oder Typo3 fallen in diese Kategorie. Diese Oberfläche erlaubt es dabei aber vornehmlich, den eigentlichen Inhalt der Seiten zu verändern, weniger das Rahmenwerk außen herum. Fortgeschrittene Webentwickler halten sich daher von den WYSIWYG-Web-Editoren fern und editieren lieber unmittelbar die in `HTML` (oder Skriptsprachen) notierten Templates.

Ein allgemein „bester“ Ansatz hat sich bis heute also nicht herausgebildet. Dennoch können die gängigen Ansätze zur Bearbeitung von Oberflächen recht gut anhand eines einzigen Merkmals kategorisiert werden: Wie weit ist die im Editor zu sehende Darstellung von der Darstellung im Browser des Endanwenders entfernt? Diese Frage ist natürlich sehr eng verknüpft mit dem fachlichen Hintergrund der Zielgruppe. Ein als „Webseitenbaukasten für Jedermann“ beworbenes Produkt wird um eine möglichst originalgetreue Darstellung im Editor bemüht sein, fortgeschrittene Entwickler nutzen dann lieber einen Texteditor mit speziellen Hilfsfunktionen wie Syntaxvervollständigung.

Bei WYSIWYG-Webseiten-Editoren handelt es sich im Normalfall um sehr spezielle Anwendungen, mit denen man allerdings sehr schnell Ergebnisse erzielen kann. Hat man eine von Ihnen verstanden, kann der Umgang mit einem konkurierenden Produkt immer noch in vielen Belangen unterschiedlich ausfallen. `HTML`-Code schaut hingegen in allen Editoren gleich aus und leidet daher weniger unter einem „Lock-in-Effekt“.

Insbesondere bei der Visualisierung von Schleifen und bedingten Ausgaben kommt es beim WYSIWYG-Ansatz systemimmanent zu Problemen. Der darzustellende Datenbestand ist im Falle von BlattWerkzeug zwar bekannt, es müssen also immerhin keine künstlichen Daten für die Vorschau fingiert werden. Dennoch muss an irgendeiner Stelle der Vorschau auch ein aktuell inaktiver Zweig einer Bedingung oder der abstrakte Schleifenrumpf mit Variablen dargestellt werden können. Endbenutzer werden hingegen niemals beide Seiten einer `if/else`-Kombination oder die „Vorlage“ für alle Schritte eines Schleifendurchlaufes zu Gesicht bekommen. Darüber hinaus gibt es in `HTML` auch noch einige unsichtbare Knoten wie zum Beispiel `<form>`, `<audio>` oder `<menu>`. Diese bedürfen einer „künstlichen“ visuellen Repräsentation, damit sie bearbeitet werden können.

Die WYSIWYG-Darstellung kann also prinzipiell nicht exakt wie beim Endanwender aussehen, sondern muss noch modifiziert werden.

Darüber hinaus bereitet auch das responsive Design der Vorschau Probleme: Es gibt schließlich nicht nur eine Art und Weise die Seite darzustellen, sondern mehrere. Zwischen diesen muss dabei zumindest umgeschaltet werden können.

Kein unlösbares, aber dennoch ein praktisches Problem bei der Implementierung von WYSIWYG-Editoren ist deren technische Komplexität. Grundsätzlich kann der Code zur Erzeugung der Darstellung für Endanwender nicht 1:1 für die editierbare Darstellung im Editor übernommen werden. Jedes Bedienelement muss im Endeffekt also doppelt implementiert werden[36]. Bei einer textuellen Repräsentation von `HTML`-Dokumenten im Editor kann die Darstellung allerdings ausgesprochen gut generalisiert werden. Schließlich handelt es sich im Kern um eine Baum-Darstellung mit einer sehr festen Struktur aus `XML`-Knoten mit Attributen.

Für einen WYSIWYG-Editor sind hingegen vielfältige Bearbeitungsmöglichkeiten erforderlich. Bilder bedürfen anderer Handgriffe als zum Beispiel Tabellen, Knöpfe oder Links. Um das System für die Entwickler der Entwicklungsumgebung wartbar und für die Lernenden verständlich zu halten, darf an dieser Stelle allerdings auch nicht blind eine spezielle Editorkomponente für jedes Bedienelement entwickelt werden. Es gilt vielmehr die möglichen Gemeinsamkeiten herauszuarbeiten und soweit wie möglich zu generalisieren. Ein dafür zu erstellendes Konzept wäre dabei sowohl technisch als auch fachlich kompliziert.

3.8.2. Vor- und Nachteile einer Blockdarstellung

Die Blockdarstellung, wie sie im Sinne dieses Kapitels verstanden wird, lehnt sich optisch an das Erscheinungsbild eines `HTML`-Text-Dokumentes an. Damit wird grundsätzlich ein ähnliches Ziel wie beim `SQL`-Editor verfolgt: Der Schritt vom BlattWerkzeug-Editor zu einem Texteditor ist nicht besonders groß, trotzdem können Syntaxfehler durch die Drag & Drop-Bedienung unterbunden werden. Und weil der Begriff „Block“ durchaus ein

[36] Im Falle von BlattWerkzeug ist das schon aufgrund der unterschiedlichen technischen Grundlagen für Server und Client notwendig (siehe Kapitel 4.1 „Client-Server-Architektur“). Allerdings wird auch in Frameworks mit nur einer Codebasis, zum Beispiel in der .net-Bibliothek WPF, im Code von Bedienelementen explizit zwischen „Designtime“ und „Runtime“ unterschieden.

```
<body >
  <heading ebene="1"> {{ query.artikel_detail.caption }} </heading>
  <paragraph >
  {{ query.artikel_detail.text }}
  </paragraph>
  <heading ebene="2"> Kommentare </heading>
  <query-table abfrage=" artikel_kommentare ">
  commenter_name  text
  </query-table>
  <form >
    <heading ebene="3"> Neuer Kommentar </heading>
    <input name=" input.name ">
    <input name=" input.text ">
  </form>
  <link ziel="Hauptseite"> Zurück zur Hauptseite </link>
</body>
```

Abbildung 25: Blockdarstellung im Editor

wenig unspezifisch sein kann, illustriert Abbildung 25 die Darstellung der Blöcke im Prototypen.

Der wesentliche Vorteil der Blockdarstellung ist Konsistenz, das gilt gleichermaßen für die Implementierung wie für die Bedienung. Anders als beim WYSIWYG-Editor müssen für neue Bedienelemente nämlich keine spezifischen Visualisierungen programmiert werden: Alles ist ein Block. Zwar unterscheiden sich einige der Attribute, die Anzahl der hierfür nötigen Sonderfälle ist aber überschaubar, zumal praktisch jede Art von Attribut bei mehr als einem Bedienelement zum Einsatz kommen kann.

Die Nähe zum Quelltext macht für Lernende den Übergang zu „normalem" `HTML` sehr einfach. Ein Blick in den Quelltext einer gerenderten Seiten, zum Beispiel mit den typischen Debuggingtools im Browser, sollte anhand von vielen bekannten Fragmenten genug Anhaltspunkte liefern, um zum Beispiel die Auswirkungen von konkreten Fallunterscheidungen oder Schleifendurchläufen nachvollziehen zu können.

Umgekehrt ist diese Nähe zum Quelltext natürlich auch der wesentliche Nachteil. Ohne eine mehr oder minder formale Einführung in `HTML`, kann man die Blöcke nur sehr mühevoll nutzen. Und anders als beim WYSIWYG-Editor kann man die Funktionsweise unbekannter Blöcke auch nur sehr eingeschränkt durch einfaches Einsetzen in das Dokument abschätzen.

Für Oberflächeneditoren jenseits von BlattWerkzeug ist das Verpacken der Quelltextfragmente in Blöcke kein sehr geläufiges Konzept. Die meisten Entwicklungsumgebungen verfolgen, wie im vorigen Kapitel angedeutet, einen hybriden Ansatz und erlauben sowohl die Bearbeitung im Quelltext als auch über einen WYSIWYG-Editor. Eine Sicht „zwischen“ diesen beiden Extremen existiert in keinem Produkt, welches dem Autor dieser Thesis bekannt wäre. Einige Umgebungen sind in der Lage, die Struktur des aktuell bearbeiteten Dokumentes als Baumdarstellung zu visualisieren, diese dient dann aber nur der schnelleren Navigation in komplexen Oberflächenbeschreibung.

Bei genauerer Betrachtung ist das aber auch nicht sehr verwunderlich: Die wesentliche Zielgruppe dieser professionellen Entwicklungsumgebungen sind erfahrene Programmierer, die selbst am besten entscheiden können, welche Visualisierung gerade sinnvoll ist. Eine Darstellung, die sich rein optisch nur sehr wenig vom Text unterscheidet, bietet da keinen besonderen Mehrwert.

Lernende sind hingegen in einer ganz anderen Situation: Sie sind mit der Gestaltung von Oberflächen wenig vertraut und wissen im Regelfall noch nicht, welche Möglichkeiten ihnen offenstehen. Optische Hervorhebungen von möglichen Optionen im Quelltext helfen dann dabei, über die nächsten Schritte zu entscheiden. Ein wesentliches Merkmal der blockigen Visualisierung ist, dass sie fehlende Parameter direkt an Ort und Stelle darstellen kann. Wenn man ein unbekanntes Bedienelement „einfach mal so“ in die Seite zieht, bekommt man also zumindestens einen Hinweis auf die konkret benötigten Parameter.

3.8.3. Mögliche Ansätze

Für BlattWerkzeug stellt sich bei der Gestaltung des Editors für Oberflächen also eine zentrale Frage: Welches Abstraktionsniveau passt zu der avisierten Zielgruppe? Und wie soll zwischen den Zielen „Motivation durch praktisch vorzeigbare Ergebnisse“ und der

Tatsache, dass BlattWerkzeug sich als Lehranwendung für `HTML` versteht, umgegangen werden?

Das Abstraktionsniveau kann dabei im Wesentlichen auf zwei voneinander weitestgehend unabhängigen Achsen variiert werden (Abbildung 26): Die horizontale Achse erstreckt sich von „Texteditor“ zu „WYSIWYG-Editor“ und beschreibt die Komplexität der Vorschau. Je näher der Editor an der textuellen Darstellung ist, desto mehr müssen die Schüler gedanklich ergänzen. Der Mittelpunkt auf dieser Achse ist ein „blockiger“ Drag & Drop-Editor bei dem mehr oder minder abstrakte Blöcke miteinander kombiniert werden. Nur in dafür besonders geeigneten Fällen erfolgen Eingaben über die Tastatur, zum Beispiel wenn es um die Eingabe von Texten für Absätze oder Überschriften geht.

Die vertikale Achse ist die Komplexität der zur Verfügung gestellten Bedienelemente, sie geht von „Reines `HTML`“ bis hin zu „Komplexe Bedienelemente“ und wurde schon in Kapitel 3.7.6 „Komplexe Bedienelemente“ diskutiert. Der Mittelpunkt dieser Achse ist eine Mischung aus `HTML`-Elementen und Liquid-Kontrollstrukturen. Unterhalb der Mitte entfällt die Nutzung des `include`-Tags für Bedienelemente, die nur mit den freigeschalteten `HTML`-Elementen nicht abgebildet werden könnten. Der Entwickler darf also keine Elemente mehr verwenden, deren Implementierung er nicht nachvollziehen könnte.

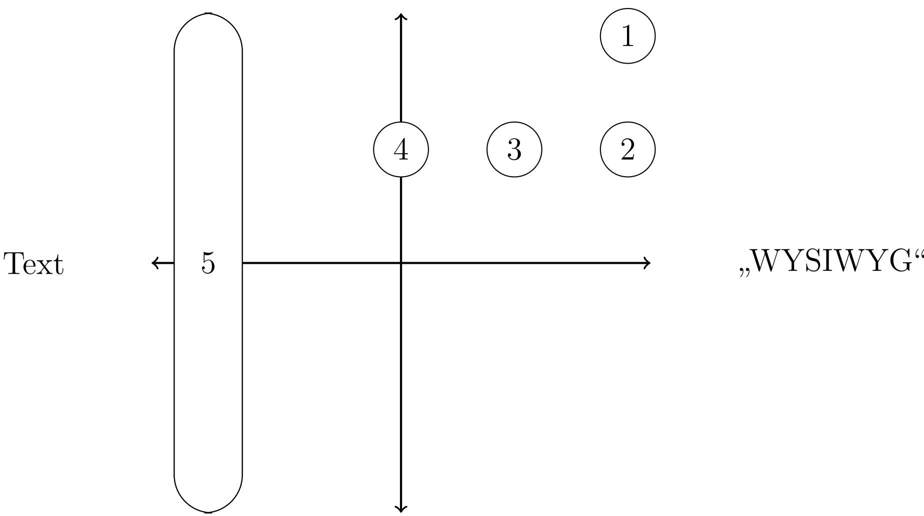
Für dieses Kapitel werden einige denkbare Ansätze auf dem beschriebenen Koordinatensystem ungefähr verordnet und deren individuelle Vor- und Nachteile diskutiert. Drag & Drop als vornehmliches Bedienparadigma ist dabei für alle Varianten gesetzt, sofern die Beschreibung dies nicht explizit abspricht.

(1) WYSIWYG-Editor für vordefinierte Seiten mit festen Einsatzzwecken

Gedacht für Entwickler, die mit BlattWerkzeug einfach nur eine generische Oberfläche für einen Datenbestand erzeugen möchten. Das Layout und die verfügbaren Bedienelemente sind dabei vordefiniert, es können lediglich an vorgesehenen Stellen Abfragen eingebunden, konstante Texte hinterlegt oder ähnliche Parametrisierungen vorgenommen werden. Prinzipiell wäre es auch denkbar, diese Angaben in einer sehr nüchternen, tabellarischen Darstellung vorzunehmen, sinnvoller erscheint im Hinblick auf den Einsatzzweck jedoch eine umfangreiche Vorschau.

Die wesentliche Herausforderung bei diesem Ansatz ist die Bereitstellung von Vorlagen, welche weder zu viele noch zu wenige Konfigurationsmöglichkeiten bieten. Ist die Vorlage zu flexibel besteht die Gefahr, dass sie für den Entwickler undurchschaubar wird. Auf der anderen Seite sind zu eingeschränkte Vorlagen aber Gift

Komplexe Bedienelemente



Reines `HTML` mit Liquid

Abbildung 26: Die horizontale Achse beschreibt die Komplexität der Visualisierung, die vertikale Achse die Komplexität der Bedienelememente

für kreative Experimente: Wenn die Antwort auf fast jede Idee der Lernenden „das geht damit nicht“ wäre, kommt schnell Frust auf.

(2) WYSIWYG-Editor mit komplexer Vorschau

Dieses Vorgehen ähnelt dem Vorgehen von Programmen wie Frontpage oder Dreamweaver. Dem Entwickler wird kein Quelltext präsentiert, sondern eine annotierte Fassung der Vorschau mit all den im vorigen Kapitel beschriebenen Problemen bezüglich der Visualisierung von Wiederholungen, Alternativen und unsichtbaren Elementen.

Kontrollstrukturen würden bei diesem Ansatz allerdings nicht durch typische Begriffe wie `for` oder `if` dargestellt, sondern durch dem konkreten Bedienelement angepasste Blöcke. Im Falle der Tabelle könnte man zum Beispiel die Spaltenüberschriften auswählbar und sortierbar gestalten. Diese Auswahl und die Reihenfolge aus der Editoransicht werden dann für die Benutzeransicht übernommen.

(3) WYSIWYG-Layout mit Block-Editor für ausgewählte Elemente

Dieser Ansatz kommt dem Entwickler noch ein Stück weiter entgegen, wenn es um

die Gestaltung des grundlegenden Layouts, also die Aufteilung in Zeilen und Spalten geht. Innerhalb der Zellen kommt dann allerdings ein an Quelltext angelehnter Block-Editor zum Einsatz. Die komplexen Bedienelemente stehen zwar weiterhin zur Verfügung, werden aber nicht mehr durch einen speziellen „Untereditor“ repräsentiert. Stattdessen werden die Parameter per Drag & Drop-Editor mit den verfügbaren Daten verbunden.

(4) Block-Editor für beliebige `HTML`- und Liquid-Blöcke

In diesem Schritt werden die Liquid-Kontrollstrukturen für Schleifen und Verzweigungen freigeschaltet. Als Entwickler könnte man jetzt eigene beliebig komplexe Strukturen bauen, auch unter Zuhilfenahme von `HTML`-Elementen wie `<div>` oder `<span>`.

(5) Texteditor für Quelltext

Da die einfachste Implementierung dieser Variante schon fast durch ein einfaches `HTML-<textarea>`-Element bereitgestellt werden kann, ist die technische Umsetzung vergleichsweise trivial. Die unterstützte Bandbreite an Bedienelementen ist grundsätzlich beliebig, Drag & Drop kommt nicht vor. Anders als bei Block- oder WYSIWYG-Editoren müssen daher keine speziellen Komponenten für einzelne Bedienelemente programmiert werden. Die textuelle Darstellung von `HTML` und Liquid fungiert als die denkbar allgemeinste Schnittstelle.

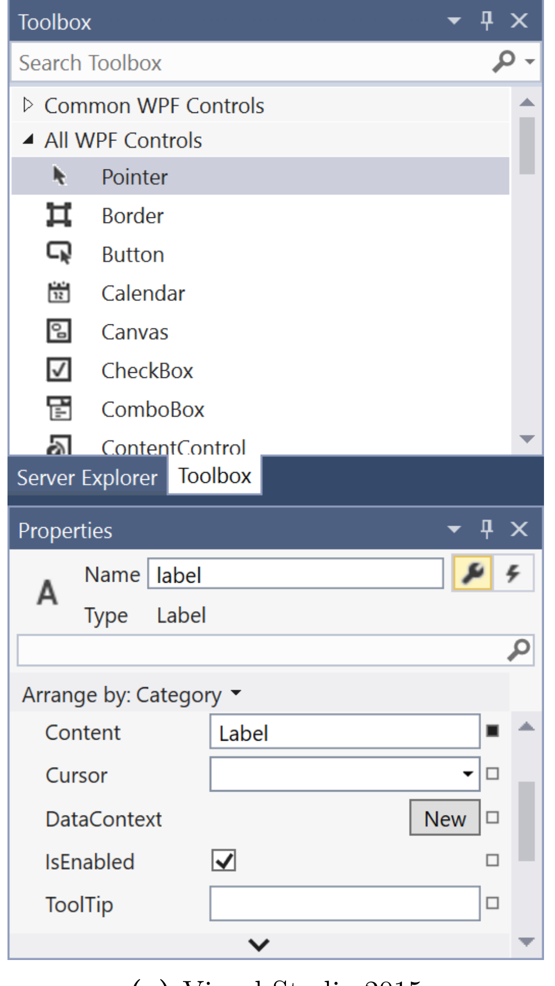
Eine detaillierte Betrachtung dieser unterschiedlichen Ansätze, also zum Beispiel die Diskussion der Erscheinungsbilder von individuellen Komponenten im Editor, wäre im Rahmen dieser Arbeit leider zu umfangreich. Außerdem ist fraglich, ob sich die Qualität von Bedienkonzepten in einer theoretischen Diskussion gut beurteilen lassen würde. Im Sinne der prototypischen Entwicklung wird im weiteren Verlauf also vor allem Wert auf die Implementierung der beiden weiter verfolgten Visualisierungskonzepte gelegt: WYSIWYG und eine Blockdarstellung.

3.8.4. Grundsätzlicher Aufbau

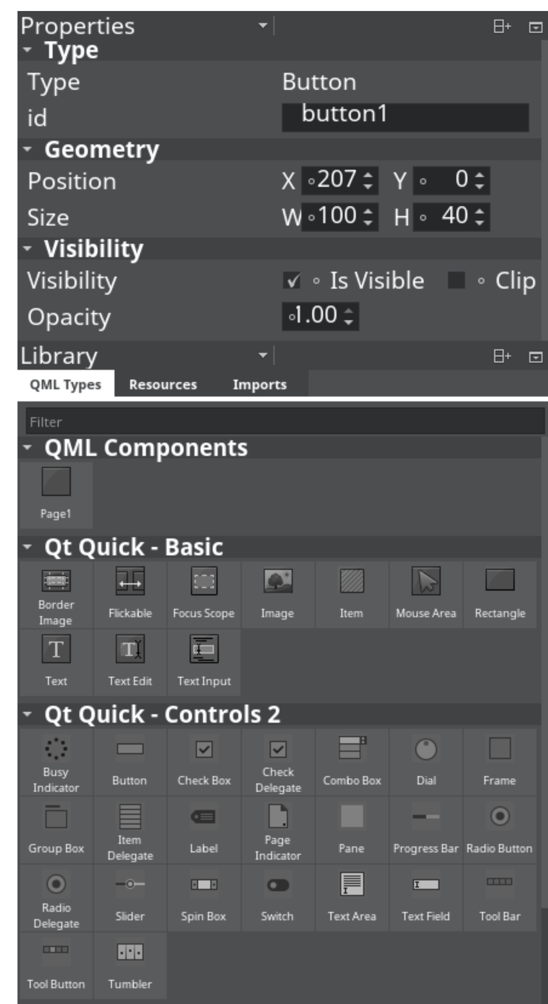
Nach dieser Betrachtung unterschiedlichster Ansätze stellt sich die Frage, inwiefern diese miteinander vereinbar sind. Dieses Kapitel stellt daher zunächst Gemeinsamkeiten heraus, für die eine Ansatz-übergreifende Lösung möglich ist. Sofern bei dem konkreten Ansatz Drag & Drop zum Einsatz kommt (mit Ausnahme des Texteditors sind das

alle) muss ein fester Bereich zur Präsentation der ziehbaren Elemente vorgesehen werden. Ebenfalls übergreifend sinnvoll ist eine generische Möglichkeit zur Bearbeitung von Eigenschaften eines Bedienelements.

Diese beiden Funktionen werden üblicherweise in eigenen Unterfenstern bereitgestellt. In dieser Form hat man dann ein sehr typisches Interface, das von so ziemlich jedem Oberflächeneditor genutzt wird. Egal ob Visual Studio, JavaFX Scenebuilder, Delphi oder QtCreator, die grundlegenden Bedienansätze unterscheiden sich an dieser Stelle nur marginal (Abbildung 27).



(a) Visual Studio 2015



(b) Qt Creator 4.1

Abbildung 27: Auswahl von Bedienelementen und Eigenschaftseditor in unterschiedlichen Entwicklungsumgebungen

Für BlattWerkzeug sollte allerdings noch ein weiterer Bereich zum Einsatz kommen: Eine

Auflistung aller verfügbaren Datenquellen. Diese Funktion wird von Entwicklungsumgebungen, im Vergleich zu dem Editor für Eigenschaften und der Aufzählung verfügbarer Bedienelemente, nur sehr uneinheitlich angeboten. Häufig liegt das an sehr großen oder nur unstrukturiert vorliegenden Datenbeständem, die manchmal sogar erst zur Laufzeit erzeugt werden. Insbesondere bei Skriptsprachen liegt der Entwicklungsumgebung in der Regel kein formales Datenmodell vor. An dieser Stelle behelfen sich die Editoren, indem sie für jede zu bearbeitende Oberfläche explizit ein formales Datenmodell einfordern. Der `WPF`-Editor des Visual Studio erlaubt zum Beispiel die Angabe einer „ViewModel"-Klasse, die dann zur Designzeit vom Editor instanziiert wird. Dementsprechend muss diese Klasse auch losgelöst von ihren typischen Datenquellen (Datenbank, Webservice, ...) agieren können.

Bei BlattWerkzeug ist der gesamte Datenbestand hingegen in Namensräumen vorstrukturiert und typischerweise in Bezug auf die Datenmenge auch übersichtlich (siehe 3.7.1 „Datenquellen für Webseiten"). Dementsprechend sollten beim Editieren der Seite die zur Verfügung stehenden Datenquellen einfach ausgewählt werden können, ohne dass dafür aus Sicht des Entwicklers besondere Vorkehrungen getroffen werden müssten.

BlattWerkzeug wird den Seiteneditor also in zwei wesentliche Bereiche unterteilen: Einen großen Hauptbereich zum Editieren und eine schmale Seitenleiste mit verfügbaren Daten, Bedienelementen und den Eigenschaften eines ausgewählten Bedienelements. Die Inhalte der Seitenleiste sollen dabei wann immer möglich zwischen dem WYSIWYG- und dem Block-Editor geteilt werden können. Die unterschiedlichen Bestandteile der Seitenleiste können bei einer größeren Anzahl von Elementen die Übersichtlichkeit negativ beeinträchtigen. Andere Entwicklungsumgebungen lösen das durch die Nutzung von Tab- oder Akkordeon-Containern, so dass sich aktuell störende Elemente ausblenden lassen. An dieser Stelle wird ein praktischer Test mit Anwendern am Besten zeigen können, welcher Ansatz für BlattWerkzeug zu bevorzugen ist.

3.8.5. Seitenleiste: Editieren von Eigenschaften

Um Eigenschaften von Bedienelementen auch ohne spezielle Komponenten editieren zu können, wird in der Seitenleiste ein Editor für Eigenschaften bereitgestellt. Diese Vorgehensweise ermöglicht eine universelle Bearbeitungsmöglichkeit unabhängig von der Repräsentation im eigentlichen Editorbereich. Sowohl ein WYSIWYG- als auch ein

„Block"-Editor können also, bewusst oder als „Fallback", auf identische generische Bearbeitungsmethoden für Bedienelemente zurückgreifen. Abbildung 28 zeigt Beispiele für die im Prototypen implementierten Editoren.

Sofern möglich werden dabei identische Eigenschaften bei unterschiedlichen Bedienelementen auch identisch angeordnet. Eingabeelemente für Formulare (28a „Eingabe" & 28b „Auswahl") fragen immer zuerst nach dem `name`-Attribut („Eingabename") und dem dargestellten Titel des editierten Elements.

3.8.6. Seitenleiste: Angebot von Bedienelementen

Technisch gesehen ist dieses Element keine besondere Herausforderung: Es handelt sich um eine einfache Auflistung von verfügbaren Bedienelementen. Um den Wiedererkennungswert von bestimmten Bedienelementen zu steigern, werden diese mit einem kleinen Symbol ausgestattet (Abbildung 29a).

Die wesentliche Herausforderung für diese Komponente liegt in der absehbar immer größer werdenden Anzahl von Bedienelementen. Der Prototyp nimmt schon den Versuch einer Kategorisierung vor (blau für Layoutelemente, grün für `HTML`-Elemente, orange für Elemente mit einer komplexen Darstellung), diese konkreten Einordnungen sind aber noch nicht endgültig.

3.8.7. Seitenleiste: Angebot von verfügbaren Daten

Die Anzeige der verfügbaren Daten ist grundsätzlich auch eine Aufzählung der Inhalte aller gefüllten Namensräume. Und ähnlich wie beim vorigen Punkt ist auch an dieser Stelle die wesentliche Herausforderung erneut die potenziell sehr große Anzahl von Bedienelementen.

Zur besseren Visualisierung sind an dieser Stelle verschiedene Symbole für die korrekte Verbindung notwendiger Parameter vorgesehen. Der in das Rechteck eintretende Pfeil (zum Beispiel bei `GET`-Parametern und Abfragen) symbolisiert Werte, die bereits vor dem Rendervorgang zur Verfügung stehen und daher in das Template eingesetzt werden können. Werte, die aus einem Rechteck austreten, stehen erst auf der Folgeseite zur Verfügung, zum Beispiel Formulardaten.

Eingabename

input.name

Titel

Spitzname

Beschreibung

Eingabetyp

Text

Pflichteingabe?

(a) Eingabe

Eingabename

input.nummer

Titel

Pokemon

Benutzte Abfrage

Pokedex_1G

Text für Option

name

Wert für Option

nummer

(b) Auswahl

Text

Zurück zur Hauptseite

Eigene Seite

Hauptseite

— ODER —

Externe URL

(c) Link

Benutzte Abfrage

Meine_Pokemon

Mögliche Spalten

gefangen_id

nummer

name

typ

spitzname

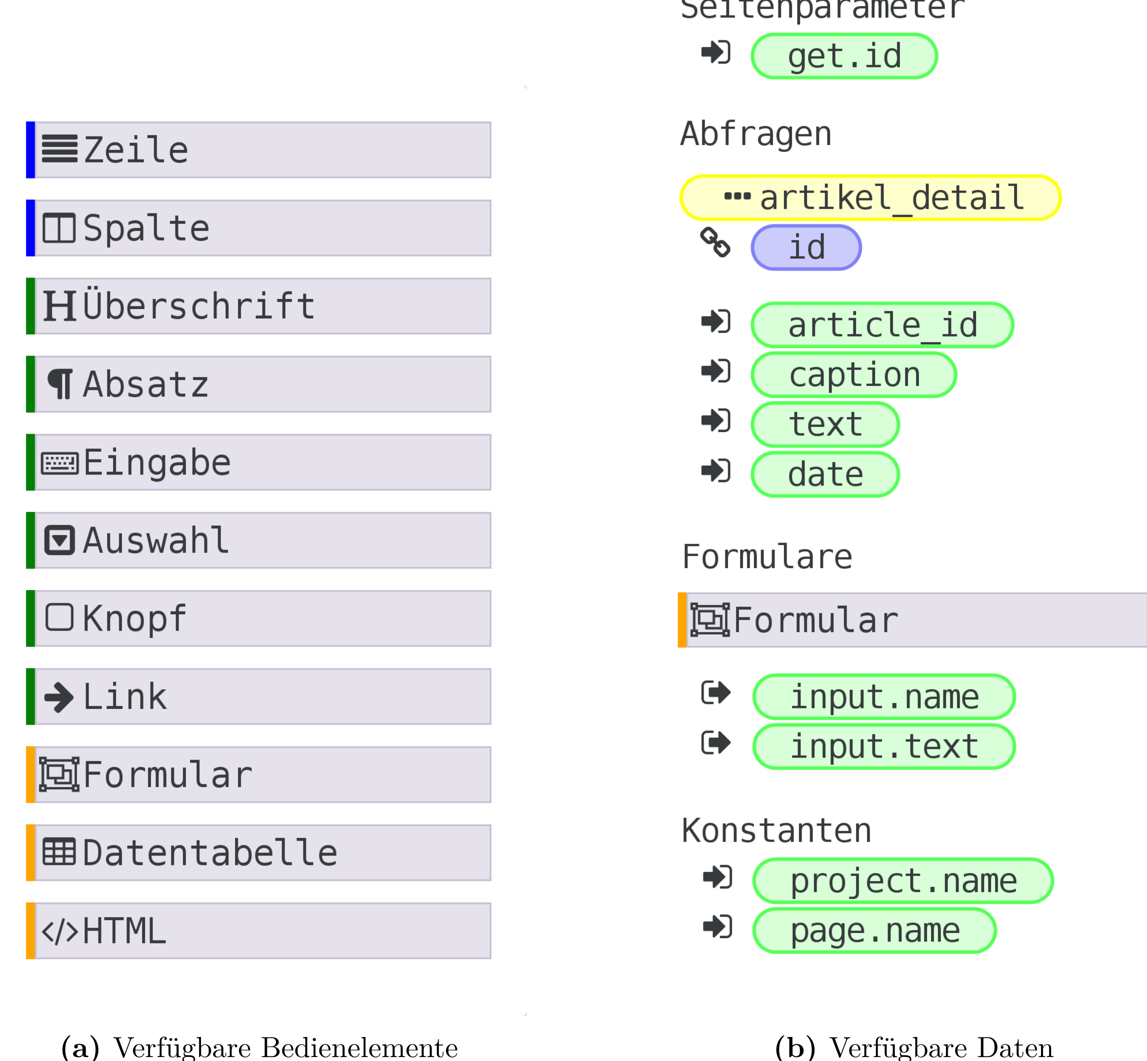
staerke

(d) Tabelle

Abbildung 28: Beispiele für Eigenschaftseditoren in der Seitenleiste

Problematisch bei dieser Visualisierung ist ein technisches Detail: Ein Formular kann in Abhängigkeit der verwendeten `<button>`-Elemente auf mehr als ein Ziel verweisen. Daher hat das `<form>`-Element intern bisher keine weitere Bedeutung außer „Elternelemente für Eingabeelemente“ und damit auch keinen Namen oder eine ähnlich identifizierende Eigenschaft, die in der Seitenleiste angezeigt werden könnte. Daher kommt es in der Darstellung zu einer Doppelung des Begriffs „Formular“.

Die Kettenglieder stehen für korrekt gebundene Parameter einer Abfrage. Sofern ein erforderlicher Parameter nicht an einen Wert gebunden wurde, verfärbt sich der Name des entsprechenden Parameters rot und das Symbol zeigt ein gerissenes Glied.



(a) Verfügbare Bedienelemente

(b) Verfügbare Daten

Abbildung 29: Weitere Elemente der Seitenleiste

3.8.8. Binden von Parametern

Die Bindung von Parametern erfolgt ebenfalls über Drag & Drop. Als Entwickler kann man ein zur Verfügung stehendes Datum von der Seitenleiste oder aus dem Editorbereich direkt auf auf den zu bindenden Parameter ziehen.

Problematisch ist dabei vor allem die in Kapitel 3.8.7 „Seitenleiste: Angebot von verfügbaren Daten“ erwähnte Unterscheidung zwischen Daten die während des Rendervorgangs nicht oder eben doch zur Verfügung stehen. Um den Wert eines Formularfeldes an einen Liquid-Ausdruck oder eine serverseitig ausgeführte Abfrage zu binden, wäre der Einsatz von Javascript nötig. Dabei handelt es sich nicht um eine willkürliche Einschränkung von BlattWerkzeug, `HTTP` und `HTML` funktioneren schlicht auf diese Art und Weise.

Den Lernenden muss daher verdeutlicht werden, zu welchem Zeitpunkt die Daten zur Verfügung stehen beziehungsweise wann diese benötigt werden. Es wäre zum Beispiel denkbar, für jedes eingesetzte Formularelement automatisch einen gleichnamigen `GET`-Parameter, gegebenenfalls mit einem Standardwert, vorzusehen. In diesem Fall stünden Formulardaten automatisch zur Verfügung, allerdings erst nach einem Klick auf den „Absenden“-Knopf.

4. Implementierung

Dieses Kapitel beschreibt den Implementierungsprozess von BlattWerkzeug und erläutert einige technische Entscheidungen. Der gesamte entstandene Quelltext steht unter der GNU Affero General Public License (AGPL)[37] und ist in einem öffentlichen Git-Repository einsehbar[38]. In diesem Repository befinden sich ebenfalls die Beispielprojekte aus Anhang A und auch der LaTeX-Quelltext zu dieser Thesis.

Als primäres Interface für die Kompilierung wird ein `Makefile` genutzt. Dieses prüft, ob auf dem aktuellen System alle nötigen Abhängigkeiten verfügbar sind und stellt sicher, dass Übersetzungsschritte nur ausgeführt werden, wenn sie tatsächlich notwendig sind. Von den technischen Details der unterschiedlichen Programmierumgebungen aufgrund der unterschiedlichen Programmiersprachen kann so außerdem elegant abstrahiert werden: Ein Programmierer kann sich die exakten Aufrufe der unterschiedlichen Paketmanager zwar anschauen, wird im Normallfall aber nur `make install-deps` aufrufen. Die `readme.md`-Datei im Repository erläutert diese Details zur Kompilierung und Inbetriebnahme ausführlich, sie sind daher nicht Bestandteil dieser Thesis.

4.1. Client-Server-Architektur

Der softwaretechnische Unterbau der Entwicklungsumgebung setzt auf aktuelle Webtechnologien auf (siehe 3.4.1 „Webanwendung“ für die Diskussion der Begründung) und teilt sich in zwei distinkte Codebasen für Server und Client.

Server: Ruby mit Sinatra

Die Aufgaben des Servers sollen sich konzeptionell möglichst auf die Auslieferung und Speicherung von Daten beschränken. Die Interaktion findet dabei primär über eine REST-artige JSON-Schnittstelle statt, serverseitig gerendert werden lediglich die Projekte der Schüler.

Client: Typescript mit Angular 2

Aufgrund des hohen Grades an Interaktivität bietet sich eine rein clientseitige Visualisierung an, die weitestgehend auf Roundtrips zum Server verzichtet. Außer

[37] `https://www.gnu.org/licenses/agpl-3.0.de.html`
[38] `http://blattwerkzeug.de/forward/git-repository`

für den Zugriff auf serverseitige Resourcen (Datenbank, gespeicherte Ressourcen, gerenderte Seiten) werden alle Operationen im Browser ausgeführt.

Um die Schnittstelle zwischen diesen beiden Komponenten so transparent wie möglich zu halten, werden diese gemäß der OpenAPI-Spezifikation[39] dokumentiert. Dieses offene Format ermöglicht es, auf eine hilfreiche Auswahl an standardisierten Tools aufzubauen. Zum Beispiel können aus der Spezifikation interaktive Testumgebungen für die Server-Schnittstellen erzeugt werden, Abbildung 30 zeigt ein Beispiel dafür.

4.2. Meilensteine

Der Editor für Datenbankabfragen versprach im Vergleich zum Seiteneditor das einfachere Teilprojekt zu sein: Die Struktur einer Abfrage ist sehr strikt festgelegt, der Umfang lässt sich gut lokal eingrenzen. Im ersten Schritt wird allerdings nicht der komplette Editor implementiert, sondern zunächst nur das interne Datenmodell für die Abfragen in Form eines abstrakten Syntaxbaumes mitsamt den zugehörigen Tests.

Um den Umfang des Prototypen zu begrenzen, wurde die Implementierungsphase anhand der folgenden funktionalen Meilensteine geplant. Jeder Meileinstein beginnt mit einer kurzen Erläuterung und listet dann die sich daraus ergebenden funktionalen Anforderungen auf:

Bearbeitung von Projektdaten

Schwerpunkt dieses Meilensteins ist die Implementierung grundlegender gemeinsamer Schnittstellen. Diese müssen vom Server als `HTTP`-Endpunkte angeboten und vom Client angesprochen werden können.

- Persistierung von Projekten mit Name und Beschreibung
- Editor für Projekteigenschaften
- Auflistung aller Projekte einer BlattWerkzeug-Instanz

Anzeige eines Datenbankschemas

Neben der funktionalen Anforderung umfasst dieser Meilenstein die Erweiterung des Clients um die typischen Bestandteile der Benutzerschnittstelle einer Entwicklungsumgebung. Serverseitig erfordert dieser Meilenstein zum ersten Mal eine Verbindung mit der Datenbank.

[39] `https://openapis.org/`

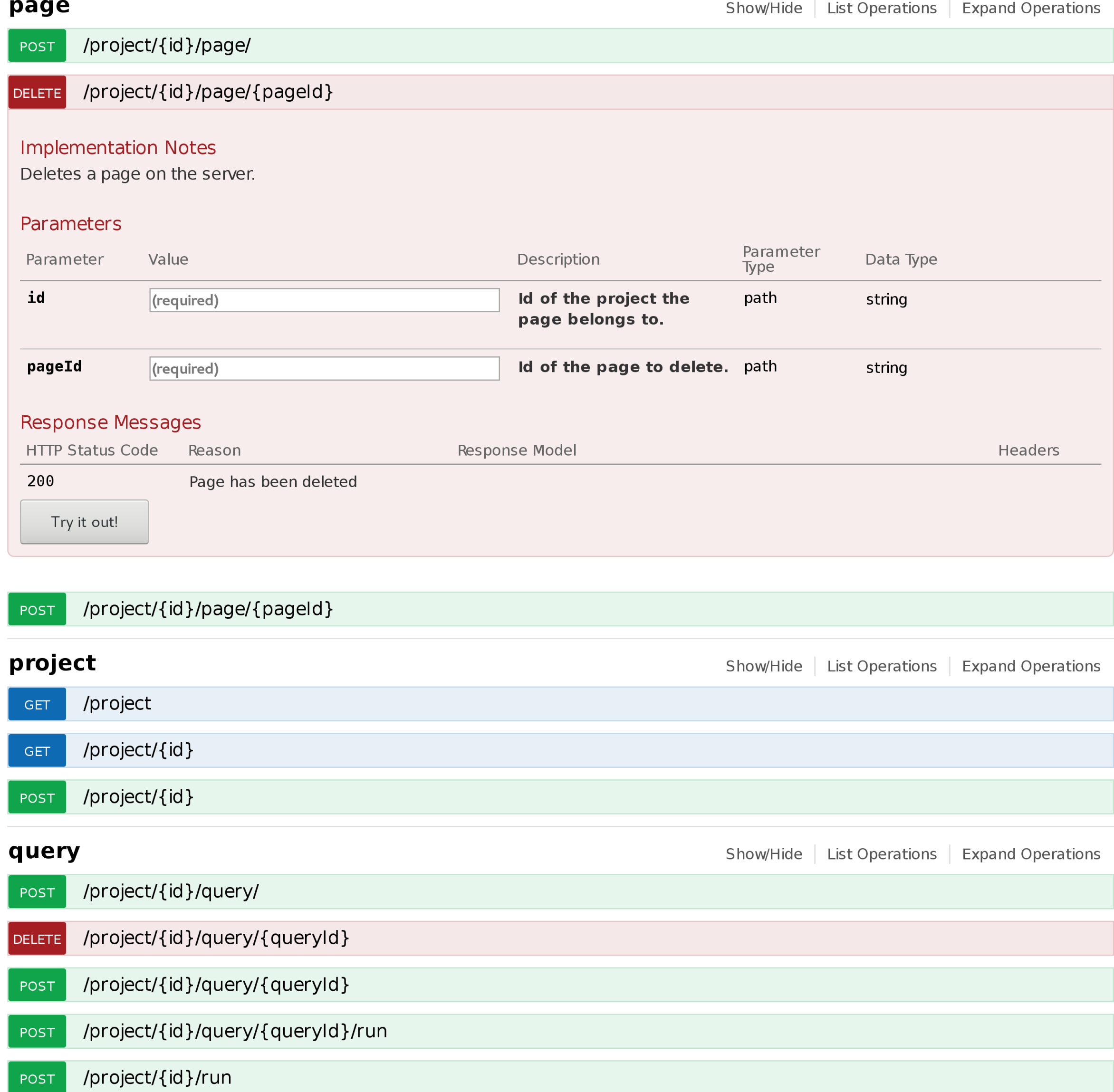


Abbildung 30: Generiert aus der Spezfikation: API-Browser für BlattWerkzeug

- Editor zur Anzeige eines Datenbankschemas
- Seitenliste mit Übersicht über alle Bestandteile eines Projekts
- Toolbar mit Knöpfen, deren Verfügbarkeit vom aktuellen Editor abhängt
- Wechsel zwischen verschiedenen Editoren

Datenmodell & Code-Generator für Abfragen

Der abstrake Syntaxbaum für Datenbankabfragen in BlattWerkzeug sowie dessen Validierung gegen ein Datenbankschema. Dieses Datenmodell soll soweit wie möglich von der Visualisierung entkoppelt werden, damit es sich möglichst einfach testen lässt.

- Einfache Projektion durch Auswahl von Spalten im `SELECT`
- Kreuzprodukte unterschiedlicher Tabellen im `FROM`
- Einschränkungen der Ergebnismenge mit Ausdrücken im `WHERE`
- Ausdrücke mit binären Operationen zwischen Spalten, Konstanten und benutzerdefinierten Werten
- Validierung des Datenmodells mit im Fehlerfall aussagekräftigen Hilfen

Editor für Abfragen

Implementierung des Drag & Drop-Editors für SQL-Abfragen. Wesentliche Herausforderung in diesem Schritt wird die Implementierung eines möglichst allgemeingültigen Ansatzes zur Behandlung der Drag & Drop-Vorgänge sein.

- Drag & Drop-Editor für Abfragen
- Ausführung von SQL-Abfragen auf dem Server
- Clientseitige Anzeige von serverseitigen Ergebnissen

Datenmodell & Code-Generator für darstellende Webseiten

Das Datenmodell für Webseiten zur Anzeige der Ergebnisse von `SELECT`-Abfragen sowie die dazu passenden Testfälle. Auch hier gilt, dass dieses Datenmodell soweit wie möglich von der Darstellungsebene (Angular 2) losgelöst sein soll.

- Layout mit Zeilen und Spalten
- Einfache, textbasierte Elemente wie Absätze und Überschriften
- Ausgabe von Abfrageergebnissen in einer Tabelle
- Navigation zu projekt-internen und externen Seiten
- Übergabe von `GET`-Parametern an Seiten

Rendern von darstellenden Webseiten

Um sicherzustellen, dass sich in dem Datenmodell für Seiten aus dem vorigen Meilenstein keine groben Schnitzer befinden, sollen diese nun den Endanwendern zugänglich gemacht werden.

- Manuelle Erstellung einiger einfacher Testprojekte mit Testdaten
- Serverseitiges Rendern der verfügbaren Testprojekte

Editor für darstellende Webseiten

Dieser Schritt profitiert hoffentlich von den Erfahrungen mit dem Drag & Drop-Editor für SQL. Anders als für den recht offensichtlich „richtigen“ Ansatz des SQL-Editors hat sich im Rahmen der Analyse für diesen Meilenstein noch keine endgültig favorisierte Darstellung gefunden. Daher ist in diesem Schritt mit mehreren Iterationen bis zur „richtigen“ Implementierung zu rechnen.

- Zuordnung von `SELECT`-Abfragen zu einer Seite
- Drag & Drop-Editor für Seitenelemente
- Integrierte Rendervorschau

Erweiterung der Seiten um Benutzereingaben

Bisher können mit Webseiten nur Informationen präsentiert, aber nicht verändert werden.

- Einführung des `<form>`-Elementes in das Datenmodell.
- Eingabe von Texten mit dem `<input>`-Element.
- *1 aus n*-Auswahl mit dem `<select>`-Element.

Qualitätssicherung

Zu diesem Zeitpunkt sollte ein voll funktionsfähiger Prototyp existieren, der jedoch noch einer rigorosen Qualitätskontrolle unterzogen werden muss. Bisher wurde der Prototyp ausschließlich durch Entwickler bedient, es stellt sich insbesondere die Frage, wie BlattWerkzeug auf Bedienfehler oder Inkonsistenzen reagiert.

- Was passiert mit bestehenden Inhalten, wenn sich das Schema verändert?
- Wie sollten Seiten gerendert werden, wenn diese fehlerhaft sind?

4.3. Spontane Ergänzungen

Sofern sich während der Entwicklung Ideen für bisher nicht bedachte Funktionalität ergaben, wurden diese in Anlehnung an den „Minus 100 Points“-Artikel von Eric Gunnerson geprüft[40]. Die folgende Idee hat es als einzige über diese Hürde geschafft:

Verwendung mehrerer Datenbanken für ein einzelnes Projekt

Zu Beginn der Entwicklung galt die Prämisse „eine Datenbank je BlattWerkzeug-Projekt“. Diese wurde geringfügig modifiziert: „Eine *aktive* Datenbank je BlattWerkzeug-Projekt“. Dabei standen zwei primäre Anwendungsfälle im Vordergrund: Zum Einen können auf diese Art und Weise Entwickler unterschiedliche Datenbestände mit identischen Abfragen oder Webseiten erproben. Zum Anderen lassen sich so vom Entwickler vorgenommene, serverseitig gespeicherte Backups einer Datenbank realisieren. Sofern bei einem Projekt mehrere Datenbanken zur Verfügung stehen, kann die aktive Datenbank über die Projekteinstellungen ausgewählt werden.

4.4. Datenbanksystem

Die Wahl des konkreten Datenbanksystems hat einen unmittelbaren Einfluss auf nahezu alle Bereiche von BlattWerkzeug. Im Einzelnen handelt es sich dabei um die exakte Variante der `SQL`-Syntax, die Rahmenbedingungen für den Betrieb der Entwicklungsumgebung und auch die Fortführung der Projekte mit externen Programmen.

Die in der Praxis häufig dominierenden Entscheidungskriterien der Skalierbarkeit, die Unterstützung unterschiedlichster Zugriffsrechte und auch die allgemeine Performance spielen nur eine sehr untergeordnete Rolle. Die zu erwartenden Datenbestände sollten normalerweise im Bereich nur einiger Megabyte liegen. Die in der Praxis vermutlich einzige relevante Unterscheidung von Zugriffsrechten wäre zwischen allgemeinem lesendem und schreibendem Zugriff, nicht jedoch auf Basis einzelner Datensätze oder komplexer Benutzergruppen. Für die Wahl des Datenbanksystems werden stattdessen die folgenden Kriterien gewählt und hinsichtlich ihrer Relevanz sortiert:

[40] `https://blogs.msdn.microsoft.com/ericgu/2004/01/12/minus-100-points/`

Kostenlose Verfügbarkeit

Der Betrieb des Datenbanksystems soll nicht mit Lizenzkosten für Schulen, Lehrkräfte, Lernende oder auch freiwillige Entwickler verbunden sein.

Einfacher Betrieb

Zwar ist für den Einsatz von BlattWerkzeug aufgrund des Browsers als Client schon die Nutzung eines Servers nötig, das Datenbanksystem sollte den Betrieb aus Sicht von Administratoren der Seite dennoch nicht mehr als unbedingt notwendig verkomplizieren. Eine wesentliche Rolle spielt dabei die Plattformunabhängigkeit: Das Datenbanksystem sollte, wie auch der BlattWerkzeug-Server, auf jeder gängigen Betriebssystemfamilie (Windows, MacOS, Linux) lauffähig sein.

Einfache Backups

Die gewünschte Exportfunktion für Projekte macht es nötig, den gesamten Datenbestand vergleichsweise einfach exportieren und importieren zu können. Darüber hinaus sollte es auch für Lehrkräfte möglichst einfach sein, mit allen Projekten zu einem anderen BlattWerkzeug-Server umzuziehen.

Tools zur Modellierung

Da diese Arbeit sich nicht mit der Datenmodellierung befasst, muss das entsprechende Datenbankschema extern erzeugt werden. Von einer guten Unterstützung für Modellierungsvorhaben profitiert dementsprechend indirekt auch BlattWerkzeug.

Externe Tools zur Entwicklung von SQL-Abfragen

Sobald ein Entwickler an die Grenzen des `SQL`-Editors von BlattWerkzeug stößt, soll es bei Bedarf so einfach wie möglich sein, die entsprechend komplizierten Abfragen in einem externen Editor zu schreiben und danach in Textform wieder an BlattWerkzeug zu übergeben.

Das Kriterium der „kostenlosen Verfügbarkeit“ ist dankenswerterweise sehr einfach zu erfüllen: Es existiert eine Vielzahl von praktisch eingesetzten quelloffenen Datenbanksystemen. Die Kriterien „einfacher Betrieb“ und „einfache Backups“ teilen die denkbaren Systeme recht eindeutig in zwei Lager: Eingebette Datenbanken lassen sich sehr einfach betreiben und sichern. Das Starten eines weiteren SQL-Server-Prozesses ist bei dieser Betriebsart nicht nötig, der Im- oder Export des gesamten Datenbestandes erfordert nur die Kopie einer einzigen Datei.

Um den Betrieb folglich so einfach wie möglich zu halten, wurden für BlattWerkzeug nur eingebettete Datenbanksysteme betrachtet. Aus der Masse an verfügbaren Systemen sticht das SQLite-System jedoch sehr weit hervor: Der Quelltext ist gemeinfrei, die Anbindung an so ziemlich jede Programmiersprache ist bequem möglich und es existiert eine Fülle von verschiedensten Modellierungsprogrammen für alle Betriebssysteme. Auf eine genauere Analyse der zur Verfügung stehenden Alternativen wurde daher verzichtet.

4.5. Tests

Die Funktionalität der relativ isolierten und daher gut zu testenden internen Datenmodelle samt den darauf definierten Operationen wird über Unit-Tests sichergestellt. Diese Tests können einfach in jedem Browser ausgeführt werden und eignen sich daher auch, um im Zweifelsfall unterschiedliche Verhaltensweisen verschiedener Browser zu erfassen.

Als technisches Fundament wird für diese Testfälle auf der Jasmine-Bibliothek aufgebaut. Zu prüfende Zusicherungen werden durch Verkettung zweier Funktionen ausgedrückt: `expect().toEqual()`. Neben `toEqual()` sind natürlich auch andere Vergleiche wie `isUndefined()` möglich. Wenn innerhalb eines Testfalls auch nur eine einzige dieser Prüfungen nicht zu `true` auswertet, wird der Testfall als ingesamt fehlgeschlagen markiert. Im Falle von mehreren gescheiterten Prüfungen werden dabei alle unerwarteten Ergebnisse aufgelistet.

Listing 21 illustriert, wie die meisten Unit-Testfälle in BlattWerkzeug aufgebaut sind. Jeder Testfall beginnt mit der Definition eines Datenmodells (Zeilen 6 bis 13) und endet mit Zusicherungen, um die korrekte Serialisierung sicherzustellen (Zeilen 21 und 22). Ganz konkret existiert also in jedem Testfall eine Variable `model`, welche im Konstruktor der zu testenden Klasse zum Einsatz kommt und in nicht-mutierenden Testfällen als Ergebnis der `toModel()`-Methode reproduziert werden muss. Für Abfragen muss noch die Generierung der korrekten `SQL`-Anweisungen geprüft werden, im Beispiel geschieht dies in Zeile 20.

Die serverseitige Funktionalität wird aktuell ausschließlich über „end-to-end"-Tests mit einem speziell instrumentierten Browser geprüft. Für diese Tests ist ein speziell vorbereitetes Testprojekt in einem exakt definierten Zustand Teil des Repositories. Auch diese Testfälle nutzen die von Jasmine bereitgestellten Zusicherungen mit den `expect()`- und

`toEqual()`-Verkettungen. Listing 22 zeigt, wie mit einem solchen Test die Funktionaliät des Editors für Projekteinstellungen sichergestellt wird. Mittels der `browser.get()`-Funktion kann zu einer bestimmten Seite navigiert werden. Um mit den einzelnen Bedienelementen dieser Seite interagieren zu können, müssen diese anhand ihrer `ID` oder anderer eindeutiger Merkmalen zugreifbar sein (Zeilen 6, 8 und 14 des Listing 22). Dann können für solche Elemente Tastatureingaben simuliert oder der Inhalt verglichen werden. Bestimmendes Merkmal dieser Tests zum Speichern ist, dass sie die initial geladene Seite nach dem Speichervorgang erneut aufrufen. Nur wenn die zuvor gesetzten Werte auch nach diesem Ladevorgang noch vorhanden sind, kann von der korrekten Funktionsweise des Servers ausgegangen werden.

Die Dateien mit den Tests liegen im Dateisystem immer „neben" ihren Implementierungen, der Dateiname wird allerdings um das Suffix `spec` wie „specification" oder `e2e` wie „end-to-end" ergänzt. Das Beispiel in Listing 21 wurde der Datei `select.spec.ts` entnommen, der Code für die zu testende Funktionalität findet sich folglich in `select.ts`.

4.6. Datenmodell

Dieses Kapitel erläutert knapp die Organisation der in BlattWerkzeug vorhandenen Entitäten. Dabei werden in den `UML`-Diagrammen auch vereinzelt Methoden aufgeführt, sofern diese die Datenstruktur verändern können, zur Serialisierung benötigt werden oder Details der Implementierung verdeutlichen. Zur Kommunikation mit der Oberfläche existieren in den meisten hier vorgestellten Strukturen noch weitere Eigenschaften, die jedoch nicht Gegenstand dieses Kapitels sind.

4.6.1. Projektressourcen

Die grundsätzliche Struktur eines BlattWerkzeug-Projektes wird in Diagramm 1 ersichtlich. Diese Darstellung visualisiert nicht die konkrete Implementierung des Servers oder des Clients, sondern illustriert die grundlegenden beteiligten Datenstrukturen. Jede dieser Entitäten - also sowohl Projekte als auch ihre Ressourcen - enthalten eine eigene Versionsangabe. Dadurch kann auf jede Veränderung an dieser Struktur explizit eingegangen werden. Aktuell laden sowohl Server als auch Client nur Ressourcen, deren Version exakt passt.

Jede Ressource (`ProjectResource`) verfügt über eine interne ID sowie einen sprechenden Namen. In der aktuellen Version von BlattWerkzeug handelt es sich bei dieser ID um eine `GUID`, sie sollte also weltweit einzigartig sein. Theoretisch wäre es dadurch denkbar, diese Ressourcen auch zwischen Projekten zu kopieren oder zu teilen. Intern werden Referenzen auf Ressourcen immer anhand der ID vorgenommen. Eine Umbenennung von Ressourcen durch den Benutzer hat daher keine Auswirkungen auf etwaige Referenzen an anderer Stelle.

Die Methode `ProjectResource.toModel()` wird genutzt, um aus dem spezifischen Ressourcen-Objekt eine rein beschreibende Datenstruktur zu erzeugen. Diese Beschreibung ist selbstverständlich abhängig vom Typ der jeweiligen Ressource und kommt auch in dessen Konstruktor zum Einsatz. Tiefe Kopien von Ressourcen können daher über diese Serialisierungs- und Deserialisierungsschritte vorgenommen werden.

4.6.2. Serverseitige Persistenz

Grundsätzlich sollte die serverseitig persistierte Repräsentation einer Ressource identisch mit dem Übertragungsformat sein. Dieses Vorgehen spart durch die Einsparung eines Transformationsvorganges potenziell Zeit, vor allem erleichtert es aber die serverseitige Implementierung. Bei der folgenden Betrachtung von möglichen Optionen zur Speicherung der Daten wird also grundsätzlich von `JSON`-Dokumenten ausgegangen. Bei der Auslieferung durch den Server können diese dann möglicherweise sogar unverändert weitergereicht werden. Um die Persistierung vorzunehmen, kommen im Wesentlichen zwei Speichermethoden in Frage: Entweder werden Datensätze direkt als Datei im Dateisystem abgelegt oder in einer dokumentenorientierten Datenbank gespeichert.

Bei einer Speicherung im Dateisystem wäre der rein lesende Zugriff auf BlattWerkzeug-Datensätze sogar mit einem einfachen Webserver für statische Dateien möglich. Um ein Projekt zu Testzwecken anzusehen oder zu editieren, reicht ein normaler Texteditor. Neue Projekte können einfach per Copy & Paste im Dateisystem angelegt werden. Perspektivisch wäre es relativ einfach möglich, ein Projekt mit gängigen Versionsverwaltungsprogrammen wie git oder mercurial zu versionieren. Diese Implementierung besticht also vor allem durch ihre Einfachheit in Bezug auf die zum Zugriff notwendigen Programme.

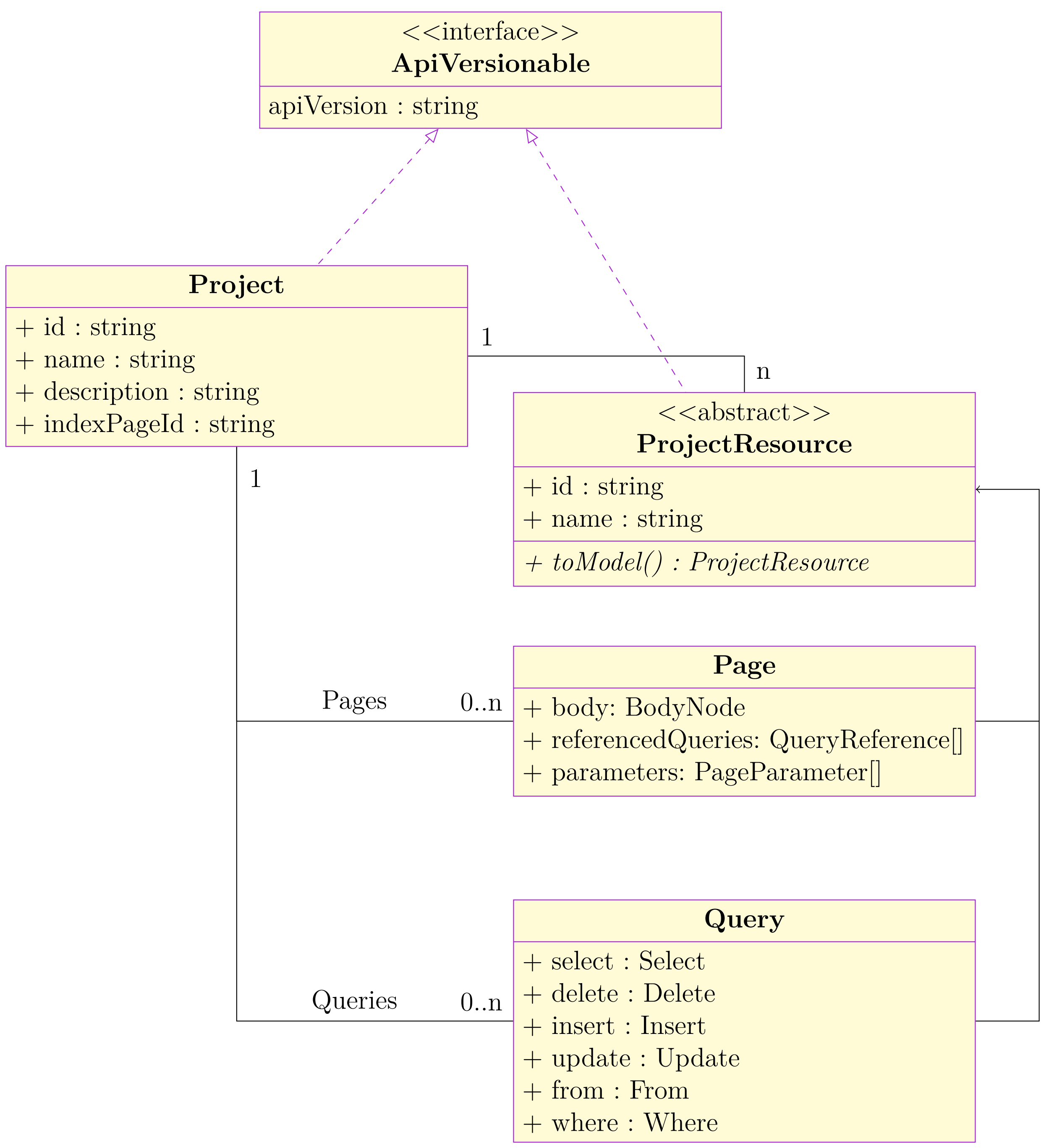


Diagramm 1: Abstrakte Übersicht über die Ressourcen eines Projektes

Die Speicherung in dokumentenorientierten Datenbanken wie MongoDB oder in einem Key/Value-Store wie redis macht den Zugriff auf die persistierten Daten abhängig von einem externen Serverprozess. Der universelle Zugriff über das Dateisystem weicht einem speziellen Programm zur Navigation innerhalb der jeweiligen Datenbank. Mit dieser Komplexität erkauft man auf der anderen Seite eine wesentlich größere Flexibilät in Bezug auf horizontale Skalierung: Sollte es jemals notwendig werden, den Server auf mehr als einem Rechner zu betreiben, existiert dafür dann schon eine hilfreiche Basis.

Letzten Endes fiel die Entscheidung auf das Dateisystem. Die Gründe ähneln jenen, wie sie schon bei der Wahl von `SQLite` aufgezählt wurden (4.4 „Datenbanksystem“): Backups des Dateisystems sind trivial, externe Programme für diesen Zweck sind zahlreich vertreten. Sollte eine komplizierte Lösung notwendig werden, kann man diese immer noch implementieren, sobald sie tatsächlich benötigt wird.

Technisch gesehen ist ein Projekt folglich eine Sammlung von Dateien in einer festgelegten Ordnerstruktur. Abbildung 31 zeigt ein beispielhaftes Projekt mit jeweils einer Datenbank, Abfrage und Webseite.

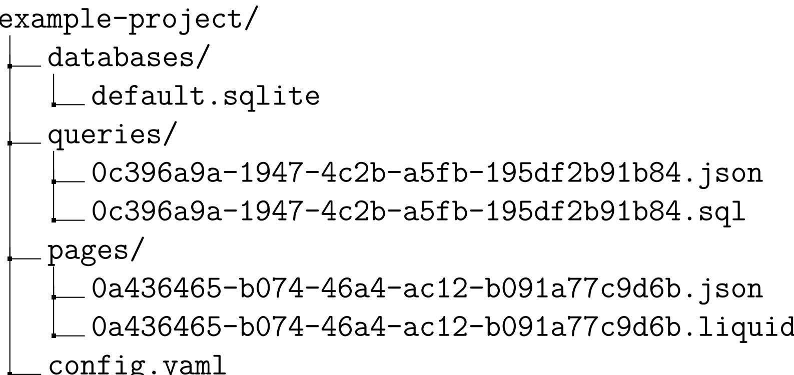


Abbildung 31: Dateistruktur eines Projekts

Dabei werden Seiten und Abfragen in zwei Versionen gespeichert: Einmal das `JSON`-serialisierte Datenmodell und eine jeweils „kompilierte“ Version in Form von `SQL`- oder Liquid-Code.

4.6.3. Projekt

Die vornehmlichen Eigenschaften eines Projektes sind die darin enthaltenen Abfragen und Webseiten, darüber hinaus werden in den Projekteinstellungen aber noch einige

andere Einstellungen festgehalten:

Name, Beschreibung & Bild
Von Entwicklern frei wählbare Elemente, mit denen das Projekt einem Endanwender kurz und knapp beschrieben wird. Der Prototyp erlaubt zum aktuellen Zeitpunkt noch nicht den Upload von Bildern, Endanwendern werden die Bilder in der Projektübersicht aber dennoch angezeigt.

Aktivierte Datenbank
Ein Entwickler kann in einem Projekt zwischen mehreren „Evolutionsstufen“ oder Sicherheitskopien einer Datenbank auswählen. Für einen Endanwender ist diese Unterscheidung allerdings nicht von Relevanz: Er erwartet nicht beim erstmaligen Besuch einer Seite zunächst eine Datenbank auswählen zu müssen. Daher legt der Entwickler in dieser Einstellung fest, welche Datenbank zum Rendern verwendet werden soll.

Startseite
Sobald in einem Projekt mehr als eine Webseite verfügbar ist, muss eine dieser Seiten als Startseite festgelegt werden.

Benutzerdatenbank
Die in Kapitel 3.4.3 „Zugriffskontrollen“ beschriebenen Zugriffskontrollen bedürfen einer rudimentären Benutzerdatenbank. Aktuell wird diese als Teil des Projekts gespeichert, die Passwörter werden dabei selbstverständlich verschlüsselt.

Routendefinitionen
Der aktuelle Stand der Implementierung erstellt für jede existierende Seite implizit eine Route, die einfach dem Namen der Seite entspricht. Eine Seite mit dem Namen „impressum“ ist folglich unter der `URL` „`http://projekt.server.tld/impressum`“ erreichbar.

4.6.4. Abfrage

Das Datenmodell für Abfragen teilt sich in drei wesentliche Bereiche auf: Zunächst die vier verschiedenen Typen von Abfragen, dann die eigentlichen `SQL`-Komponenten und zuletzt die in fast allen Komponenten denkbaren Ausdrücke. Dabei wird strikt zwischen den unterschiedlichen Typen von Abfragen (`SELECT`, `UPDATE`, `INSERT`, `DELETE`) und deren implementierenden Komponenten getrennt. Diese Trennung dient der einfachen Einbin-

dung von mehrfach auftretenden Komponenten wie `WHERE` in unterschiedlichen Abfragetypen.

Die einzelnen Komponenten der Abfragen erben von einer gemeinsamen Basisklasse `Component`. Jede konkrete Komponente wird dann als eigene Klasse implementiert, Diagramm 2 zeigt einen Ausschnitt der an `SELECT`-Abfragen beteiligten Datenstrukturen. Die Methoden `toModel()` und `toSqlString()` dienen der Überführung des Syntaxbaumes in die Repräsentation für die `.json`- beziehungsweise die `.sql`-Dateien.

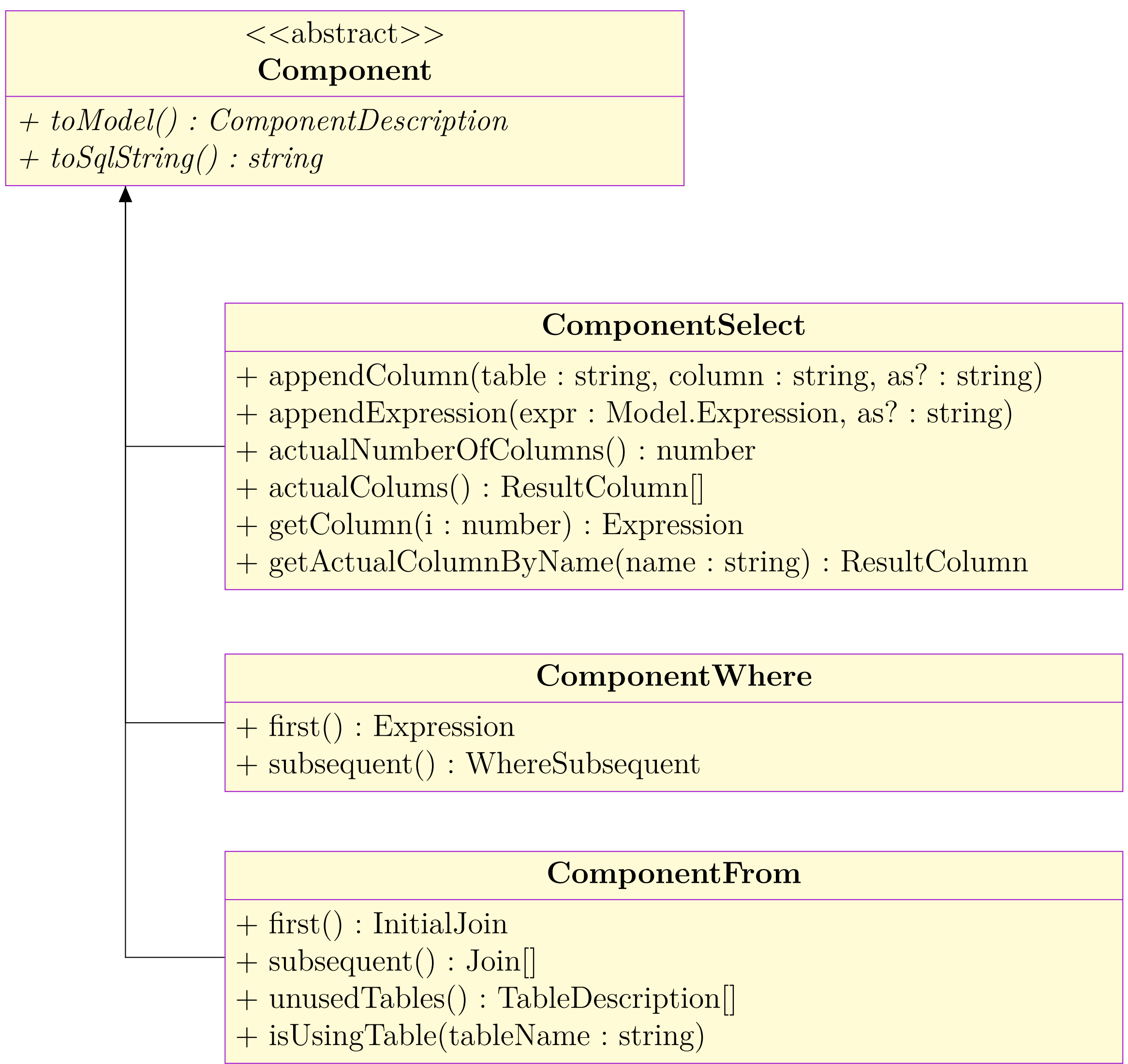


Diagramm 2: Bestandteile des Syntaxbaums für `SQL`-Komponenten

In der `SelectComponent` fällt bei den Methoden die Differenzierung in „normale“ und „tatsächliche“ Spalten auf, diese Unterscheidung ist dem `*`-Operator geschuldet. Wenn

dieser beim Zugriff auf Spalten expandiert werden soll, werden die resultierenden Spalten als „tatsächliche“ bezeichnet. Bei den „normalen“ Spalten handelt es sich hingegen um die im `SELECT` vorkommenden Ausdrücke.

Sowohl `WhereComponent` als auch `FromComponent` dürfen nicht leer sein, müssen also in jedem Fall einen Ausdruck oder einen `JOIN` bereitstellen. Ohne diese Einschränkung könnte zum Beispiel eine Abfrage ohne `WHERE`-Komponente auf zwei Arten dargestellt werden: durch eine `null`-Referenz in der Abfrage selbst oder eine `null`-Referenz als Ergebnis der `first()`-Methode. Die auf dieses initiale Element folgenden Tabellen oder Ausdrücke können mit der Methode `subsequent()` erfragt werden.

An der Wurzel des Modells stehen die konkreten Klassen für die vier verschiedenen Typen von Abfragen insgesamt, diese erben alle von einer abstrakten Basisklasse `Query`. Diagramm 3 illustriert die Hierarchie exemplarisch an dem Datenmodell für `SELECT`-Abfragen. Die Verbindung von den Abfragen zu den implementierenden Komponenten wird über eine einfache Aggregation bereitgestellt. Die konkrete Klasse `QuerySelect` stellt die Komponenten dann zur Verfügung.

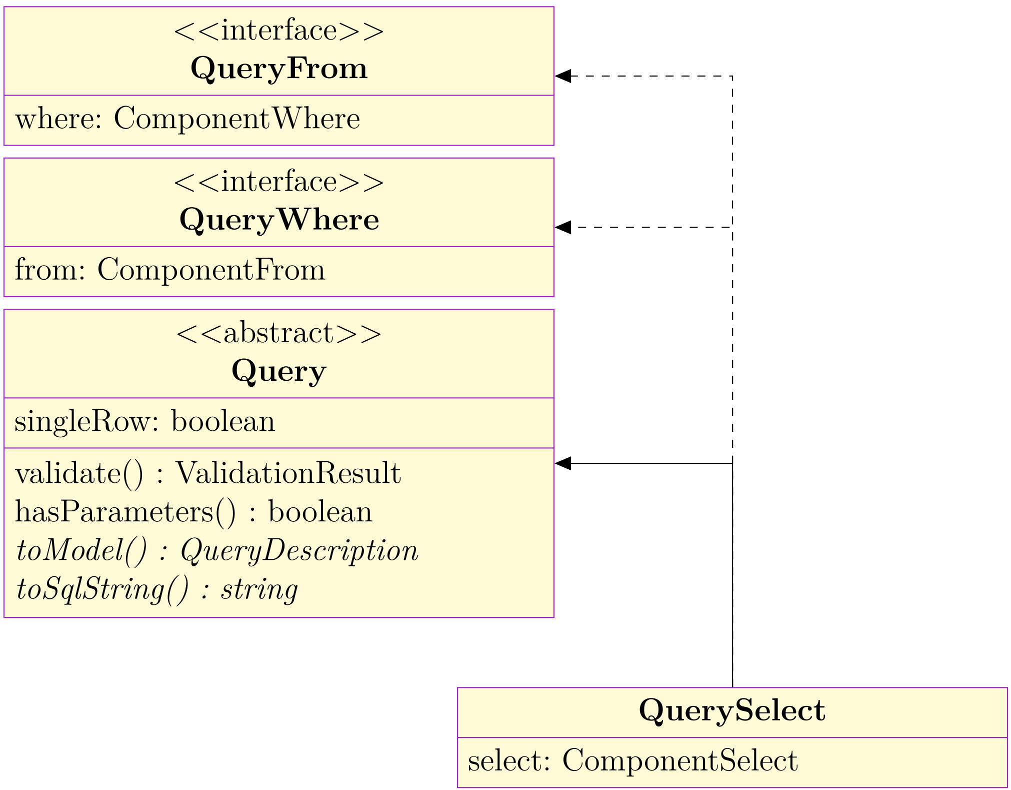


Diagramm 3: Bestandteile des Syntaxbaums für die `SQL-SELECT`-Abfrage

Die Ausdrücke werden ebenfalls über eine Klassenhierarchie definiert, an deren Wurzel allerdings unterschiedliche Klassen stehen können: jede Implementierung des Interfaces `ExpressionParent`. Dieses wird einerseits von der Basisklasse `Expression` implementiert, andererseits aber auch von jeder mit Ausdrücken arbeitenden Komponenente. Für einen Syntaxbaum ungewöhnlich ist die Implementierung von mutierenden Operationen auf den Knoten selbst. Jede Instanz von `ExpressionParent` muss in der Lage sein, Kind-Ausdrücke zu entfernen oder zu ersetzen.

Konkret existierende Ausdrücke erben dann unmittelbar von `Expression`. Diagramm 4 illustriert die vorhandenen Implementierungen samt der relevanten Daten. Leider erfordert diese gewählte Hierarchie die Implementierung von semantisch unsinnigen Operationen für logische Blätter des Syntaxbaumes: Auch eine Klasse wie `ConstantExpression` muss `replaceChild()` formal implementieren, praktisch erfolgt dies mittels einer zu werfenden Ausnahme.

4.6.5. Schwächen des Modells für Abfragen

Rückblickend könnte diese Hierarchie vereinfacht werden. Die vier konkreten `Query`-Klassen ließen sich relativ problemlos zu einer einzigen Klasse zusammenfassen. Die feine Unterteilung wurde von der Visualisierung getrieben: Eine Klasse zur Visualisierung der `WHERE`-Komponente sollte nicht irgendeine Abfrage entgegennehmen, sondern nur Instanzen die eine solche Komponente auch bereitstellen könnten. Da die meisten Komponenten aber optional sind oder über Abhängigkeiten verfügen (kein `HAVING` ohne `GROUP BY`) scheint das Typsystem von Typescript der falsche Ort, um die Einhaltung dieser Restriktionen zu prüfen.

Diese Fehlentscheidung hat leider auch die Oberfläche beeinflusst: Um eine neue Abfrage anzulegen, muss der Entwickler schon bei der Erstellung den Typ mit angeben. Würde man die Definition der Natur einer Abfrage einfach umdrehen, also anstatt einer konkreten Klasse die zugeteilten Komponenten nutzen und den abstrakten Typ dann einfach mit einer Methode ermitteln, könnte auf diese vorzeitige Angabe verzichtet werden.

Für die Klassenhierarchie der Ausdrücke wäre es auf lange Sicht sinnvoll, die Implementierung von `ExpressionParent` aus der Basisklasse `Expression` zu entfernen. Im Nachhinein ist auch dem Autor dieser Arbeit nicht mehr ganz klar, wie es zu dieser

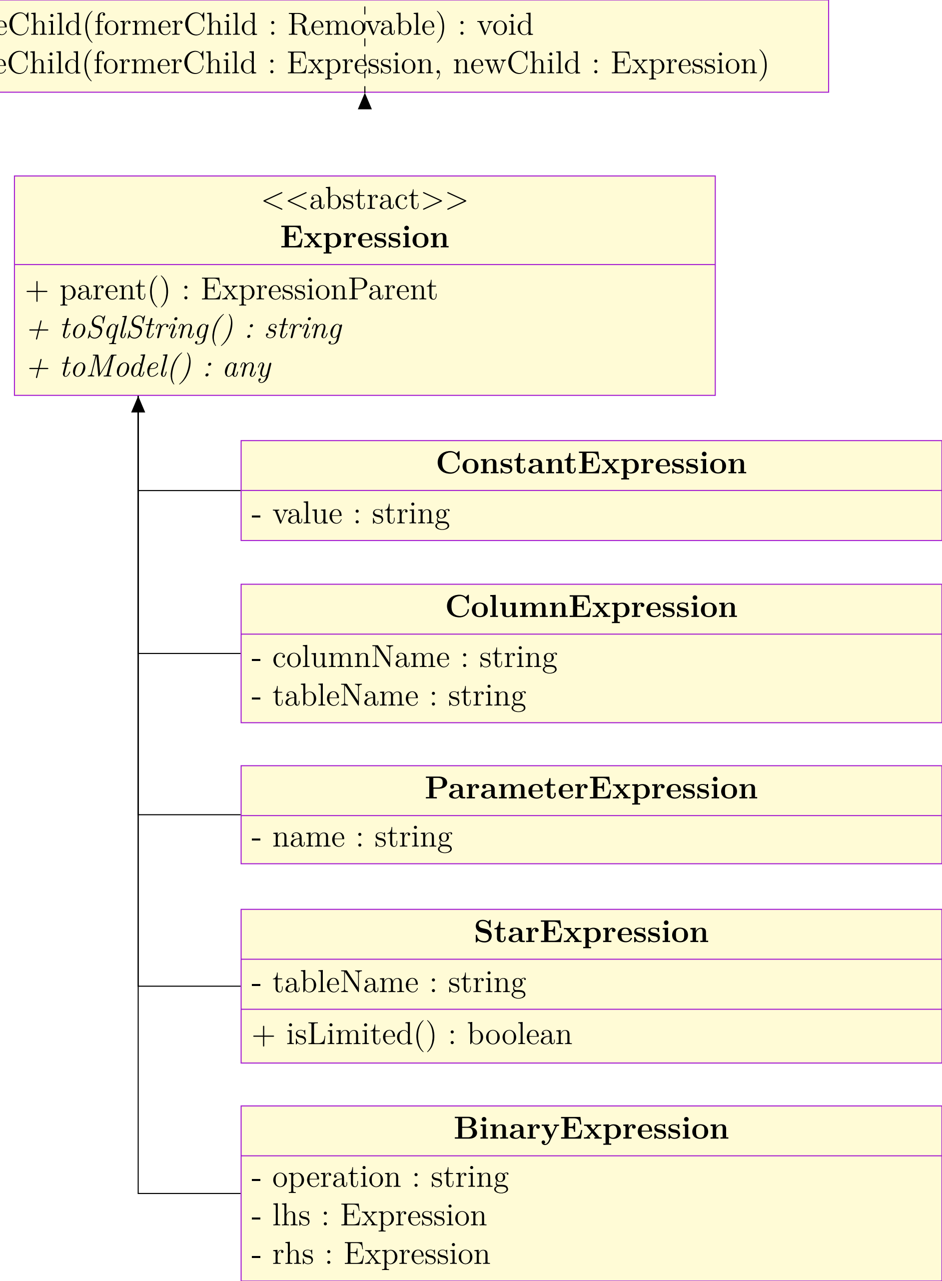


Diagramm 4: Syntaxbaum für Ausdrücke

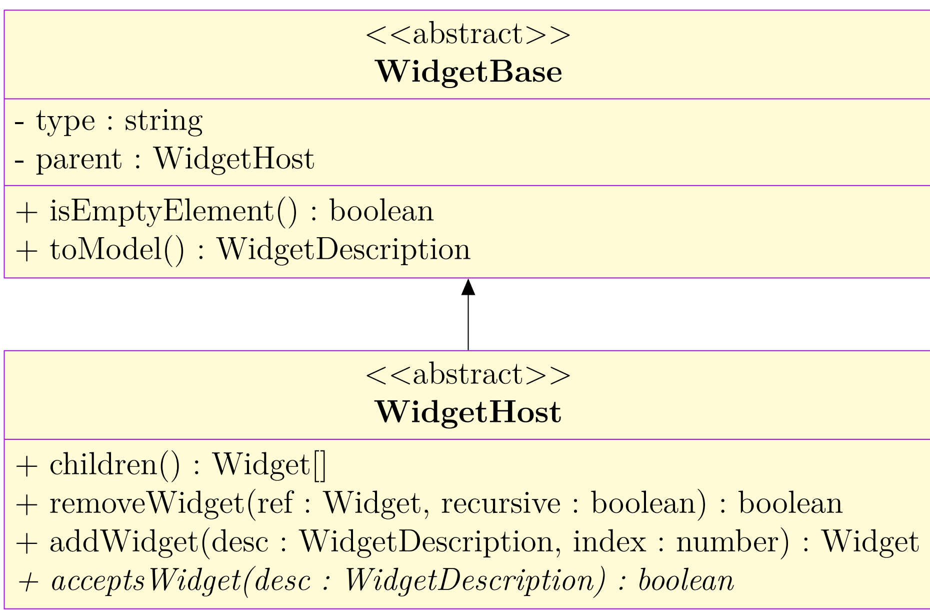


Diagramm 5: Datenmodell für Bedienelemente

semantisch unsinnigen Beziehung gekommen ist, eine Behebung war in der Kürze der Zeit aber leider nicht mehr möglich.

4.6.6. Seite

Das Datenmodell der Seite ist dem Baum des Document-Object-Model (`DOM`) nachempfunden. Die sich ergebende Hierarchie geht folglich von einem einzigen Wurzelelement aus, was in der konkreten Implementierung die Klasse `Page` ist. Diese verfügt dann wiederum über Instanzen der Klassen `Body` und `Head`.

Alle Bedienelemente erben von der abstrakten Basisklasse `WidgetBase`, für Containerelemente kommt eine Klasse namens `WidgetHost` zum Einsatz. Eigenschaften wie eine Liste von angewendeten `CSS`-Klassen oder das `id`-Attribut könnten im Hinblick auf eine `CSS`-Integration noch in `WidgetBase` ergänzt werden.

Die konkreten Bedienelemente müssen im Gegensatz zum Datenmodell für Abfragen keine `toHtml()` oder eine ähnlich benannte Serialisierungsmethode implementieren. Die Kompilierung des Datenmodells für Webseiten erfolgt mittels einer `Renderer`-Instanz. Dieser Indirektschritt wurde eingebaut, um im Falle eines Falles mehr als eine `HTML`-Templatingsprache zu unterstützen.

4.7. Client

Der Code für den Client folgt den Angular 2-Konventionen und gliedert sich auf oberster Ebene in unterschiedliche Module, die jeweils für einen spezifischen Teil einer angefragten `URL` zuständig sind. Diese Module gliedern sich dann im Wesentlichen in so genannte „Komponenten“ und „Services“. Komponenten fügen sich wie normale `HTML`-Knoten in den Quelltext ein, die Parametriesierung erfolgt über Attribute. Services dienen hingegen dem Datenaustausch und haben keine visuelle Repräsentation.

4.7.1. Kommunikation von Veränderungen

Angular 2 macht ausgiebigen Gebrauch von dem Gedankenmodell der ReactiveX-Bibliothek[41]. Der Kern dieses Modells besteht darin, eine Variable nicht als einen atomaren Wert aufzufassen. Stattdessen wird eine Variable als Serie von veränderlichen Werten mit einer festen Reihenfolge betrachtet. Die Tagline der Bibliothek lautet dementsprechend „The Observer pattern done right“. Werte, die innerhalb der BlattWerkzeug-Oberfläche verfügbar sein sollen, werden daher als ein so genanntes `Observable` zur Verfügung gestellt. Auf solcherart observierbaren Instanzen lassen sich dann eine Reihe von mutierenden oder filternden Operationen definieren. Das Verarbeitungsmodell ist dabei oberflächlich dem von `UNIX`-Pipes sehr ähnlich.

Um zum Beispiel zu verhindern, dass die Oberfläche sich bei sehr kleinteiligen Änderungen häufig ändern muss, kann der `debounce`-Filter eingesetzt werden. Dieser verzögert alle eingehenden Elemente um eine frei definierbare Zeitspanne und gibt sie nach dieser Wartezeit aus, sofern keine erneuten Werte eingetroffen sind (Abbildung 32).

Manchmal kann es auch notwendig sein, die eingehenden Werte zu transformieren. BlattWerkzeug macht von dieser Möglichkeit ausgiebig Gebrauch, um die Ergebnisse von `HTTP`-Operationen aus einer `JSON`-Darstellung in die interne Repräsentation zu wandeln. An dieser Stelle kann dann die aus der funktionalen Programmierung bekannte `map`-Funktion eingesetzt werden (Abbildung 33).

Die wesentliche Stärker dieser (und natürlich auch vieler weiterer) Operationen liegt vor allem in ihrer Kombinierbarkeit. So ist es mit den beiden vorgestellten Operatoren ein leichtes, die Veränderungen einer Abfrage leicht zeitlich verzögert (`debounce`) in einer

[41] `http://reactivex.io/`

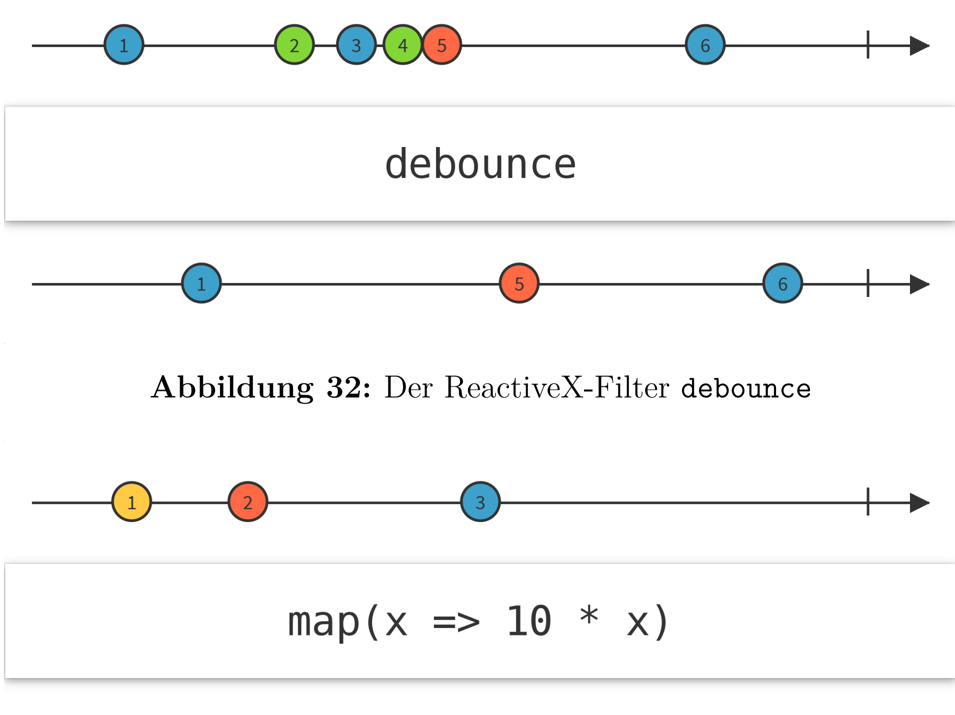


Abbildung 32: Der ReactiveX-Filter `debounce`

Abbildung 33: Der ReactiveX-Transformator `map`

textuellen Repräsentation (`map` mit `toSqlString`) auf der Oberfläche anzuzeigen. Der BlattWerkzeug-Kern nutzt daher die ReactiveX-Erweiterungen in Form der Javascript-Bibliothek „RxJS“[42].

4.7.2. Existierende Implementierungen von Drag & Drop-Editoren

Die Drag & Drop-Editoren sind allesamt von Grund auf vorgenommene Eigenentwicklungen, welche direkt auf der `HTML5`-Spezifikation aufsetzen. Im Rahmen des schon vorgestellten App Inventor (2.2 „Software: App Inventor“) wurde von Google allerdings ebenfalls eine quelloffene Bibliothek für Drag & Drop-Editoren namens „Blockly“ entwickelt[43]. Warum wurde für BlattWerkzeug also eine Eigenentwicklung forciert?

Um Blockly zu evaluieren ist ein sehr kleiner Prototyp zur Bearbeitung von `SQL`-Abfragen entwickelt worden (Abbildung 34). Dabei hat sich kein schlagender Grund gegen Block-

[42] `https://github.com/Reactive-Extensions/RxJS`
[43] `https://developers.google.com/blockly/`

ly gefunden, stattdesen aber eine recht lange Liste von technischen und inhaltlichen Aspekten, die nicht so recht zu BlattWerkzeug passen:

- Es handelt sich bei Blockly um eine Bibliothek für imperative Programmiersprachen, inklusive Code-Generatoren für zum Beispiel Python, Javascript, Lua, ... Eine Unterstützung von `HTML` oder `SQL` ist nicht vorgesehen und müsste selber implementiert werden.
- Code-Elemente werden in Blockly standardmäßig frei auf einer zweidimensionalen Zeichenfläche platziert. Um innerhalb eines Kontextes auf Ereignisse zu reagieren ist dieser Ansatz sinnvoll. Für BlattWerkzeug gilt jedoch, dass jedes zu bearbeitende Element feste Einstiegspunkte hat: `SELECT` und `FROM` bei lesenden `SQL`-Abfragen, `<body>` und möglicherweise `<head>` für Webseiten. Die mehrfache Verwendung dieser Wurzelelemente ist inhaltlich nicht sinnvoll, eine freie Platzierung bietet keinen signifikanten Mehrwert.
- Das von Blockly verwendete Datenmodell enthält Details zur grafischen Darstellung und eignet sich nicht gut zur abstrakten Repräsentation von `HTML` oder `SQL`. Dieses Format kommt daher im Hinblick auf andere Arten von Editoren, zum Beispiel einem `WYSIWYG`-Editor für Oberflächen, nicht als exklusives Format in Frage. Dementsprechend müsste immer zwischen den eigentlichen BlattWerkzeug-Formaten und Blockly hin- und zurück-übersetzt werden.
- Blockly versteht sich nicht nur als Oberflächenbibliothek, sondern auch als Code-Generator. Dieser Aspekt wird in BlattWerkzeug aber losgelöst von der grafischen Repräsentationen durch einzelne Editoren betrachtet. Diese Funktionalität von Blockly müsste daher aktiv ignoriert werden, da einzelne Schnittstellen das Vorhandensein entsprechender Funktionalität voraussetzen.
- Blockly geht zunächst von einem statisch großen Zeichenbereich aus. Um diesen in der Größe zu verändern, muss auf globale `onResize`-Ereignisse reagiert werden. BlattWerkzeug kommt hingegen mit ausschließlich per `CSS` gehandhabten Größenangeben aus, was im Endeffekt permanent korrekte Größenverhältnisse garantiert. Die offizielle „Resizeable Blockly“-Demo[44] lässt sich hingegen durch Nutzung der Entwicklerfunktionen zur Bildschirmgrößen-Vorschau in Chrome und Firefox aus dem Tritt bringen.

Da BlattWerkzeug zumindest perspektivisch die Verwendung von unterschiedlichen Editoren für identische Inhalte vorsieht, ist Blockly damit keinesfalls endgültig aus der Welt.

[44] `https://blockly-demo.appspot.com/static/demos/resizable/overlay.html`

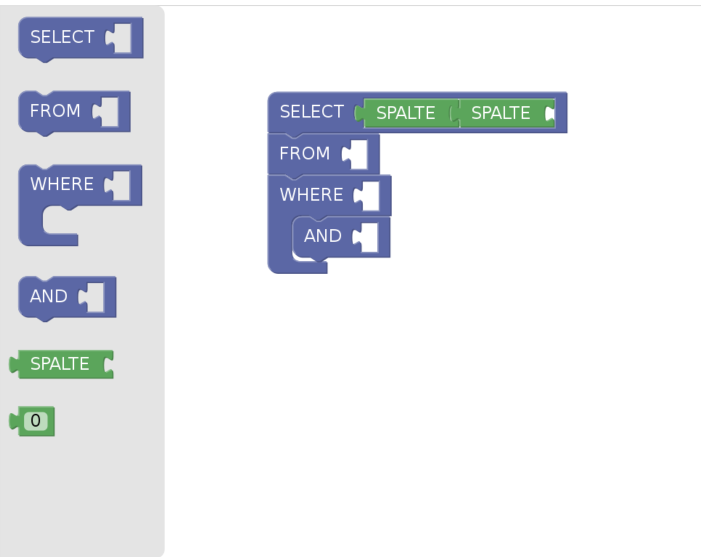


Abbildung 34: Grundzüge eines `SQL`-Editors mit Blockly

Spätestens bei einer hypothetischen Einbindung von „echten" Programmiersprachen in BlattWerkzeug wird eine Integration wieder zur Diskussion stehen.

4.8. Server

Der Server beschränkt sich auf drei wesentliche Aufgaben und ist dementsprechend von überschaubarem Umfang:

1. Bereitstellung der statischen `HTML`- und (aus Typescript kompilierten) Javascript-Dateien, sofern kein „normaler" Webserver vorgeschaltet wird.
2. Entgegennehmen von Speicheranfragen und Prüfung auf deren Plausibilität beziehungsweise Befugnis.
3. Rendern der Webseiten für Endbenutzer.

Die ersten beiden dieser Punkte werden unmittelbar von Sinatra unterstützt. Die Bereitstellung von statischen Daten funktioniert über die `send_file(path)`-Methode, Authentifizierungen können über einen entsprechenden `HTTP`-Auth-Header angefordert werden. Um die eingehenden Daten zu validieren, werden alle Anfragen gegen ein `JSON`-Schema validiert. Das Rendern der Seiten für Endbenutzer ist hingegen etwas komplizierter, die Rahmenbedingungen dafür werden daher in diesem Kapitel etwas ausführlicher betrachtet.

4.8.1. Nutzung von Subdomains

BlattWerkzeug-Projekte werden unter jeweils eigenen Subdomains zur Verfügung gestellt. Das erhöht zwar die Anforderungen an die technische Infrastruktur, geht jedoch mit einer Reihe von Vorteilen einher. Die online bereitgestellte Fassung des Protoypen geht in Bezug auf die zur Verfügung stehenden Subdomains noch einen Schritt weiter: Die Domain für die Entwicklungsumgebung (`blattwerkzeug.de`) unterscheidet sich von jener für die Projekte (zum Beispiel `cyoa.blattzeug.de`).

Der hauptsächliche Grund für dieses Vorgehen ist die im Internet weitestgehend gültige Annahme, dass auf jeder (Sub-)Domain nur ein logischer Internetauftritt hinterlegt ist. Dass sich hinter einer `URL` wie `blattwerkzeug.de/project/pokemongo` eine völlig andere Seite als hinter `blattwerkzeug.de/project/cyoa` verbirgt, ist zwar in keinster Weise verboten, aber zumindest ungewöhnlich.

Praktisch werden Subdomains von vielen Suchmaschinen, sonstigen Angeboten im Internet und auch innerhalb der technischen Standards anders behandelt, als wenn die gleichen Inhalte als untergeordnete `URL` einer anderen Seite existierten. Das beginnt schon bei entscheidenden technischen Details: So können zum Beispiel Zertifikate für `SSL`-Verbindungen nur für Domains, nicht aber für Teile einer `URL` ausgestellt werden. Und Webseiten werden im Falle von unerwünschten Inhalten, zum Beispiel Phishing-Angriffen, üblicherweise auf Basis von Domainnamen gesperrt.

Darüber hinaus kann unter dieser Voraussetzung jede Webseite davon ausgehen, dass die gesamte Routing-Hierarchie nur ihr gehört. Die Subdomains erlauben also die Nutzung von absoluten Pfadangaben. Insbesondere um Bilder, `CSS`-Dateien oder JavaScript-Bibliotheken zu verlinken, ist das eine hilfreiche Garantie, welche viele unpraktische Probleme vermeidet.

4.8.2. Rendervorgang

Der wesentliche Inhalt des Rendervorgangs besteht im Sammeln der für die Anzeige notwendigen Daten, wie sie in Kapitel 3.7.1 „Datenquellen für Webseiten“ beschrieben worden sind. Ein Großteil dieser Daten kann unmittelbar aus dem Datenmodell herausgelesen werden: Die Ruby-Klassen für Projekte und Seiten verfügen über dezidierte Me-

thoden zur Bereitstellung der Daten ihres Namensraumes (Listing 23). Die `GET`-Daten werden unmittelbar von dem serverseitigen Framework zur Verfügung gestellt.

Um die Abfragen in das richtige Format zu bringen, ist hingegen ein wenig Aufwand erforderlich: Jede in einer Seite referenzierte Abfrage muss ausgeführt werden. Dieser Prozess erfolgt in zwei Schritten.

Zunächst werden die verfügbaren Parameter auf die tatsächlich benötigten Parameter abgebildet. Zur Erinnerung sei daher an dieser Stelle noch einmal die grundsätzliche Problematik erwähnt: Die Daten für die Seiten liegen in Namensräumen vor, die Parameter einer Abfrage wissen hingegen nichts von diesen Namensräumen. Listing 24 zeigt, wie die einzelnen Mapping-Einträge einer referenzierten Abfrage in Namensraum und Schlüssel aufgespalten und dann unter dem in der Abfrage definierten Parameternamen verfügbar gemacht werden.

Nachdem die Parameter dann übersetzt worden sind, erfolgt die eigentliche Ausführung der Abfrage (Listing 25). Dabei werden die als `string[][]`-Array zurückgelieferten Rohdaten in ein Array von Hash-Maps übersetzt (`Dict<string, string>[]`). Nach dieser Präparation der Parameter und der Transformation der erhaltenen Zeilen kann das Ergebnis der Abfrage dann in die Renderdaten eingepflegt werden.

Der eigentliche Rendervorgang wird von der Liquid-Bibliothek durchgeführt. Deren Interface ist aber wie in Kapitel 3.7.2 „Evaluation existierender Templatingsprachen“ beschrieben denkbar einfach: Übergeben wird ein String mit dem zu rendernden Template und die in diesem Kapitel beschriebene `HashMap` mit den zur Verfügung stehenden Daten.

4.9. Unerwartete Hindernisse

Im Laufe der BlattWerkzeug-Entwicklung traten, vornehmlich bei der Entwicklung des Webclients, einige unerwartete Hindernisse auf. Jene Probleme, die subjektiv betrachtet wesentlich mehr Zeit gekostet haben als eingeplant war, werden an dieser Stelle kurz erwähnt.

4.9.1. Instabile API von Angular 2

Zum Start der Entwicklung von BlattWerkzeug befand sich Angular 2 gerade in der Version `2.0.0-beta.0`. Auf diese initiale Version folgten 17 weitere Beta-Veröffentlichungen und 7 „Release-Candidates“. Alle in diesen Veröffentlichungen getätigten „Breaking Changes“ mussten von BlattWerkzeug mitgemacht und nachvollzogen werden. Zu den besonders schwerwiegenden Umbauten gehören dabei die folgenden Anlässe:

- Das interne Routing-Modell wurde zweimal komplett umgeschrieben, dabei waren jedes Mal manuelle Änderungen an den Routendefinitionen nötig[45]. Zum aktuellen Zeitpunkt liegt der Router daher schon in Version 3 vor, während Angular 2 noch auf Version 2 verweilt.
- Die interne Darstellung von Abhängigkeiten einzelner Komponenten wurde mit dem Release-Kandidaten 5 komplett überarbeitet. Bisher hatten Komponenten Abhängigkeiten selber referenziert, nun sollen diese dafür in ein neu entwickeltes Modulsystem einsortiert werden.
- Die Syntax der Templatingsprache wurde in Bezug auf die Angabe von Referenzen verändert. Dabei wurden Schlüsselwörter verändert und gestrichen, was eine komplette Durchsicht aller Template-Dateien erforderte.

4.9.2. Fehlerhafte Codegenerierung des Typescript-Compilers

Der verwendete Typescript-Compiler hat zum Zeitpunkt der Anfertigung dieser Arbeit einen bekannten Bug in der Codegenerierung[46], um den wiederholt herumgearbeitet werden musste. Konkret äußert sich dieser Fehler, wenn die Definition der Oberklasse einer sich davon ableitenden Klasse erst im Nachhinein erfolgt (Listing 26). In diesem Fall kommt es zu keiner Warnung durch den Compiler, sondern zu einem Laufzeitfehler im kompilierten Javascript-Code. Das Beispiel ist an dieser Stelle natürlich besonders plakativ, der Fehler tritt aber in identischer Art und Weise bei komplexen Hierarchien aus `import`-Anweisungen auf.

[45] Siehe `http://stackoverflow.com/questions/38039294/` und `http://stackoverflow.com/questions/38879529/#38952908`

[46] `https://github.com/Microsoft/TypeScript/issues/4341`

4.9.3. Verworfene Implementierung eines reinen WYSIWYG-Seiten-Editors

Vor dem im aktuellen Prototypen verwendeten Block-Editor (zu sehen auf zum Beispiel Abbildung 25 oder im Anhang A „Projektbeispiele“) wurde noch ein reiner WYSIWYG-Editor entwickelt. Dieser ist in der Lage, beliebige Layouts aus Zeilen und Spalten mit einer akkuraten Vorschau zu versehen. Der nächste Schritt in der Implementierung wäre neben der Vorschau von Knöpfen und Eingabefeldern die Darstellung der Datentabellen gewesen (Abbildung 35).

Die Weiterentwicklung wurde dann jedoch nach einem Gespräch mit Dr. Frank Huch in einer gemeinsamen Entscheidung gestoppt: Dieser Editor wäre eine weitere Insellösung gewesen, mit der die Lernenden zwar hätten schnell erste Erfolge vorweisen können, dabei die technischen Hintergründe jedoch nicht verstanden hätten. So weit sollte die „Motivation durch praktisch vorzeigbare Ergebnisse“ dann doch nicht getrieben werden, schließlich handelt es sich bei BlattWerkzeug in erster Linie immer noch um ein Lehrprogramm. Vollständig Vergebens waren die investierte Zeit dennoch nicht: Das entwickelte Datenmodell für die Seiten konnte beibehalten werden, der serverseitige Code zum Rendern ebenfalls.

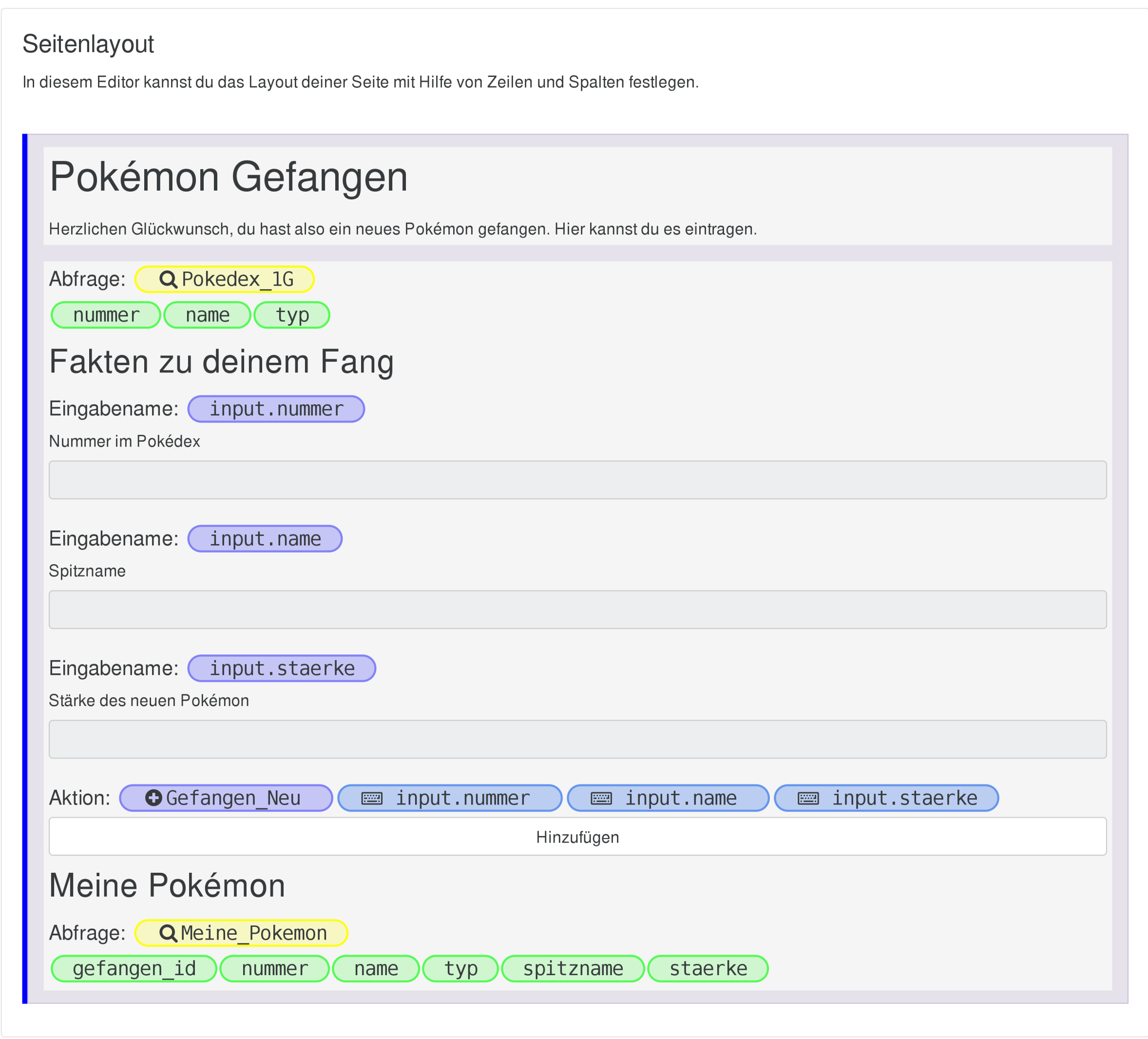


Abbildung 35: Entwickleransicht des verworfenen WYSIWYG-Seiten-Editors

```
// Datamodel for "SELECT * FROM person WHERE 1"
// Table "person":
//   - Primary Key "personId", int
//   - Column "name", string
//   - Column "gebDat", int
const model : Model.QueryDescription = {
    name : 'where-simple',
    id : 'where-1',
    apiVersion : CURRENT_API_VERSION,
    select : { columns : [{ expr : { star : { } } }] },
    from : { first : { name : "person" } },
    where : { first : { constant : { value : "1" } } }
};
let q = new QuerySelect(schema, model);
// SELECT, star needs to be expanded according to the model
expect(q.select.actualColums).toEqual(3);
// FROM, has no joins at all
expect(q.from.numberOfJoins).toEqual(0);
// Serialization to SQL and the internal model
expect(q.toSqlString()).toEqual('SELECT *\nFROM person\nWHERE 1');
expect(q.toModel()).toEqual(model);
```

Listing 21: Unit-Test für eine korrekte `SELECT`-Abfrage

```
1  import { browser, element, by } from 'protractor'
2
3  describe('Test Project: Settings', () => {
4    const settingsUrl = `/editor/test/settings`;
5    it('can be edited & saved', () => {
6      browser.get(settingsUrl);
7      // Random name & description
8      const nameEle = element(by.id('project-name'));
9      const nameVal = Math.random().toString(36).substr(2);
10     const descEle = element(by.id('project-description'));
11     const descVal = Math.random().toString(36).substr(2);
12     // Setting it
13     nameEle.clear()
14       .then( () => nameEle.sendKeys("CTRL+a"), nameVal))
15       .then( () => descEle.sendKeys("CTRL+a"), descVal))
16       .then( () => element(by.id('toolbar-btn-save')).click())
17       .then( () => {
18         // Reload the page, asserting that everything has been saved
19         browser.get(settingsUrl);
20         // Ensure everything has been saved
21         expect(nameEle.getAttribute("value")).toEqual(nameVal, "Name
              differed");
22         expect(descEle.getAttribute("value")).toEqual(descVal, "
              Description differed");
23     });
24   });
25 });
```

Listing 22: End-to-end Test für den Speichervorgang eines Projektes

```
# Retrieves the "site" hash that may be used during rendering.
#
# @return [Hash] Rendering-relevant data for a project
def render_params
  return {
    'name' => self.whole_description['name'],
    'id' => self.id,
    'description' => self.whole_description['description'],
  }
end
```

Listing 23: Bereitstellung der Renderdaten in der Projektklasse

```
# Each query gets its own fresh set of parameters
params = {}

# Not-quite-so-wellformed models may omit the mapping, this
# shouldn't crash anything immediatly.
query_ref.fetch('mapping', {}).each do |mapping|
  # Extract all relevant indizes
  prov_prefix, prov_name = mapping.fetch('providingName').split "."
  parameter_name = mapping.fetch('parameterName')

  # And do the actual mapping
  mapped_value = initial_params
                 .fetch(prov_prefix)
                 .fetch(prov_name)
  params[parameter_name] = mapped_value
end
```

Listing 24: Renderdaten einer Abfrage (1): Parameter binden

```
# Grab the actual query and execute it with the constructed parameters
query = @project.query_by_id(query_ref['queryId'])
result = @project.execute_sql(query.sql, params)

# Templating engines works much better with 'sensible' keys as names,
# so we map the column names into each row. This basically transforms
# rows like [1,2,3] to { "column-name-1" => 1, ... }
mapped = result['rows'].map { |r| Hash[result['columns'].zip r] }

# Rows with a single value allow a short-hand notation. The user
# shouldn't be forced to write {{ row[0].column }} every time he
# **knows** there is only a single row.
if query.single_row?
  if mapped.length == 1
    mapped = mapped.first
  else
    err_msg = "Got #{mapped.length} rows, expected exactly 1"
    raise DatabaseQueryError.new(@project, query.sql, params, err_msg)
  end
end

return (mapped);
```

Listing 25: Renderdaten einer Abfrage (2): Ausführen und transformieren

```
class B extends A {
    constructor(msg:string) {
        super(msg);
    }
}

class A {
    constructor(public msg:string) {
    }
}
```

Listing 26: Falsche Reihenfolge der Klassendefinition

5. Fazit

Ein Blick auf die im Anhang vorgestellten Projekte (A „Projektbeispiele“) zeigt, dass der Prototyp dem eingangs erwähnten Ziel[47] durchaus gerecht wird: Zu praktisch beliebigen SQLite-Datenbanken können Abfragen und Oberflächen entwickelt werden.

Etwas detaillierter lässt sich der Grad des Erfolges einer Software vor allem am finalen Entwicklungsstand festmachen: Welcher prozentuale Anteil der angestrebten Funktionalität wurde erreicht? Daher muss sich der „fertige Prototyp“ an den in Kapitel 3.2 „Grundprinzipien“ formulierten Zielen messen lassen. Auch einige spätere Kapitel, insbesondere 3.5 „Konzept für sinnvolle Teilmengen von `SQL`“, lassen sich als ein recht umfassender Anforderungskatalog verstehen. Die formulierte Roadmap (siehe 4.2 „Meilensteine“) wurde bis auf den letzten Punkt „Qulitätssicherung“ eingehalten. Eine Ausweitung der in Kapitel 4.5 „Tests“ beschriebenen Testmethodiken steht also ebenfalls noch aus.

Dieses vermeintlich objektive Kriterium der prozentualen Vollständigkeit berücksichtigt aber nur einen Teil der für Prototypen wie BlattWerkzeug relevanten Ziele. Da in diesem Fall auf dem prototypischen Stand weiter aufgebaut werden soll, ist eine weitere Frage von Bedeutung: Kann man auf diesem Stand die Weiterentwicklung fortführen?

5.1. Erreichte Ziele

Der aktuelle Stand der Implementierung ist vor allem als ein Durchstich zu verstehen: Auch wenn es in fast jedem Teilbereich noch vereinzelt an Funktionalitäten fehlt, kann das Zusammenspiel dieser Systeme schon gut erprobt werden. Insbesondere bei der Verbindung der Datenbank mit der Oberfläche, sowohl lesend als auch schreibend, haben sich im Laufe der konkreten Implementierung noch einige unerwartete Stolpersteine aufgetan (4.9 „Unerwartete Hindernisse“). Das zugrundeliegende Fundament, also die internen und textuellen Darstellungen von `SQL` und `HTML`, sind aber stabil und darüber hinaus auch mächtiger, als der mit dem Drag & Drop-Editor editierbare Stand.

[47] „Mit dem Blattwerkzeug lassen sich gestützt durch *Drag & Drop-Editoren* für beliebige `SQLite`-Datenbanken *Abfragen formulieren* und *Oberflächen entwickeln*“, siehe Kapitel 1 „Einleitung“

Das Grundprinzip „**Semantik vor Syntax**“ wurde erreicht, der Drag & Drop-Editor schließt Syntaxfehler kategorisch aus. Sofern diese im kompilierten Quelltext doch auftreten, wäre das eindeutig ein Fehler in der Codegenerierung, nicht jedoch des Entwicklers. Bei Fehlermeldungen für erkannte Fehlersituationen gibt es jedoch noch Verbesserungspotenzial: Aktuell werden fehlerhafte Eingaben einfach zugelassen und dann erst im Nachhinein als Fehler markiert. An dieser Stelle wäre das im Prinzip angesprochene „kontinuierliche Feedback“ vermutlich am besten mit einer eingebauten, kontextsensitiven Hilfe unterstützen. Diese könnte auf Fehler mit einer kurzen Erläuterung reagieren und dabei demonstrieren, wie die richtige Vorgehensweise wäre.

Ob die erstellbaren Seiten durch „**praktisch vorzeigbare Ergebnisse motivieren**“, kann nur ein Test mit Probanden der Zielgruppe zeigen (siehe 3.1 „Zielgruppe“). Ein Fortschritt gegenüber der nicht sonderlich hohen Hürde „besser sein als Texteingaben in einer `SQL-IDE`“ ist aber durchaus zu erwarten.

Die „**einfache Inbetriebnahme**“ ist vor allem eine Frage der Perspektive. Aus Sicht eines Schülers ist der Aufruf einer `URL` in der Tat einfach. Bisher hat BlattWerkzeug unter allen ad-hoc probierten Kombinationen aus Betriebssystemen (Windows, MacOS, Linux) und Browsern (Firefox, Chrome, Edge) gut funktioniert. Für Lehrkräfte wird aktuell eine virtuelle Maschine bereitgestellt. Der Umgang mit dieser ist aufgrund des fehlenden Webinterfaces allerdings noch relativ unbequem.

Eine Notwendigkeit des Wechsels auf eine „normale“ Desktopanwendung hat sich zu keinem Zeitpunkt ergeben. Das technische Fundament aus Ruby mit Sinatra und Typescript mit Angular 2 hat sich ebenso bewährt, wie die Entscheidung, den größten Teil der Logik im Client zu belassen. Der Betrieb von BlattWerkzeug ist nach der initialen Ladezeit fast vollständig frei von Verzögerungen. Die Möglichkeit der Entwicklung von Unit- und End-to-End-Tests fügt sich gut in den Entwicklungszyklus ein.

5.2. Nicht erreichte Ziele

Das Ziel einer „**schrittweise komplexeren Benutzeroberfläche**“ wurde zunächst hinten angestellt, da es mit einem ausgearbeiteten Konzept einhergehen sollte. Kapitel 3.5.3 „Sprachstufen“ untersucht zwar die möglichen Einschränkungen für `SQL`, geht aber nicht auf `HTML` oder die möglichen Auswirkungen auf das Zusammenspiel beider Systeme

ein. Mit dem aktuell noch überschaubaren Funktionsumfang ist der akute Bedarf nach diesem spezifischen Grundprinzip allerdings auch noch nicht gegeben.

Zudem wurde das Ziel der „**Fortführung der entwickelten Projekte**“ mit externen Programmen im Rahmen des Prototypen hinten angestellt. Zumindest die Unterstützung der Quelltext-Editoren sollte noch implementiert werden, bevor der Bedarf für einen Export überhaupt abgeschätzt werden kann.

Der Umfang der tatsächlich implementierten `SQL`-Funktionen ist noch nicht abgeschlossen. Zum jetzigen Zeitpunkt fehlen neben der Unterstützung der `AS`-Direktive im Editor (der Syntaxbaum unterstützt die Benennung bereits) vor allem Funktionen und Gruppierungen. Dieser Umstand schränkt die umsetzbaren Projekte empfindlich ein: Stünden diese Funktionen schon zur Verfügung, könnten zum Beispiel auch Webseiten für Sportvereine, inklusive der dynamischen Erzeugung von Tabellen, mit BlattWerkzeug umgesetzt werden.

Der Einsatz des Prototypen im Unterricht ist aktuell vor allem aufgrund der rudimentären Benutzerverwaltung nur schwer vorstellbar. Noch ist keine Registrierung von Benutzern implementiert, außerdem können Projekte nicht über die Webseite kopiert werden. Die Implementierung einer Benutzerverwaltung ist allerdings eine vor allem handwerkliche Aufgabe und wurde daher im Rahmen dieser Thesis nicht vorangetrieben.

Aber auch der Betrieb für Lehrkräfte gestaltet sich deutlich komplizierter als erwünscht: Die Notwendigkeit einer eigenen (Sub-)Domain ist keine triviale Hürde. Die Annahme „man wird doch wohl mal auf dem Nameserver der Schuldomain einen Eintrag für BlattWerkzeug anlegen können“ hat sich zwar nicht als haltlos erwiesen, behindert die initiale Inbetriebnahme aber merklich.

5.3. Weiterentwicklung

Eine offene Frage bei der Weiterentwicklung von BlattWerkzeug ist die Wahl des Zeitpunkts, zu welchem die eigentliche Zielgruppe (sowie deren Lehrkräfte) in die Entwicklung mit einbezogen werden. Oder anders ausgedrückt: Es stellt sich die Frage nach dem minimalen Satz an implementierten Funktionen, mit denen man sinnvoll Feedback bei Lehrern und Schülern einholen kann.

Die folgende Aufzählung von offenen Baustellen gibt vor allem Anhaltspunkte, an welchen Stellen die prototypische Implementierung unvollständig ist. Diese Punkte sind dabei notwendig für einen tatsächlichen Einsatz im Unterricht, aber nicht zwingend hinreichend. Insbesondere die Erfüllung von weichen Kriterien, zum Beispiel eine umfassende Qualitätssicherung oder bessere Dokumentation für Schüler & Lehrkräfte, werden nicht aufgeführt.

Allgemein: Textuelle Editoren

Momentan begrenzen die Drag & Drop-Editoren den Leistungsumfang ganz erheblich: Das interne Datenmodell ist hingegen darauf ausgelegt, auch textuelle Darstellungen „blind" entgegenzunehmen. Um Frust aufgrund von (noch?) nicht unterstützten Ideen zu vermeiden, sollten daher Texteditoren als „Notausgang" implementiert werden. Im Falle von `HTML` ist das über das spezielle Code-Bedienelement schon in Ansätzen möglich, für `SQL` hingegen noch überhaupt nicht.

SQL: Unterstützung für Funktionen und `GROUP BY`

Ohne Aggregation von Daten lassen sich viele alltägliche Fragestellungen nicht beantworten. Glücklicherweise ist die eigentliche Implementierung nicht besonders aufwändig: Der Syntaxbaum ist darauf vorbereitet, für die Oberfläche ergeben sich keine bisher nicht schon gesehenen Anforderungen.

SQL: Schemaeditor

Die Umsetzung eigener Ideen ist mit dem gegenwärtigen Stand faktisch nicht durchführbar, da es keine Möglichkeit gibt, das Datenbankschema über die Weboberfläche zu editieren. Eine Minimallösung wäre der einfache Upload von SQLite-Dateien, dann erfordert aber jede Anpassung des Schemas den Wechsel in ein externes Werkzeug.

Die optimale Lösung wäre ein in BlattWerkzeug integrierter Schema-Editor. Mit diesem sollte man mindestens Tabellen und Spalten anlegen, umbenennen sowie löschen können. Aufwändig, aber sehr interessant wäre noch die Definition von Beziehungen.

Seiten: Drag & Drop für interpolierte Ausdrücke

Gegenwärtig müssen Liquid-Objekte unmittelbar in geschweiften Klammern im Quelltext notiert werden, es findet nicht einmal Syntax-Highlighting statt. Dieser Stand bricht daher nicht nur mit dem Drag & Drop-Paradigma, er verwirrt auch mit schwer nachvollziehbaren Meldungen bei Syntaxfehlern. Folglich sollte in einer zukünftigen Version auch für diesen Bereich Drag & Drop-Unterstützung durch BlattWerkzeug erfolgen.

Seiten: Drag & Drop für Kontrollstrukturen

Bisher ist der Entwickler darauf angewiesen, dass für Wiederholungen spezielle Bedienelemente wie die Datentabelle zur Verfügung gestellt werden, Verzweigungen lassen sich momentan außerhalb des `HTML`-Text-Elementes überhaupt nicht nutzen. Um diesem Umstand zu begegnen, sollten die Liquid-Tags `for` und `if` wie die vorhandenen Bedienelemente im Drag & Drop-Editor repräsentiert werden können.

Und abseits von den Details der Editoren für `HTML` und `SQL` ist auch noch eine Grundsatzentscheidung zu treffen: Die Form, in der BlattWerkzeug Lehrkräften zugänglich gemacht werden soll. Das Feedback aus informellen Gesprächen gibt zumindest Anlass, die eingangs formulierte Prämisse „jeder Lehrer stellt halt einen Server für seine Klassen bereit“ erneut zu hinterfragen. Ein dazu häufig formulierter Kritikpunkt lässt sich mit „Immerhin wird Scratch auch einfach so im Web angeboten“ zusammenfassen. Dieser Ansatz wäre natürlich auch denkbar, eine solche Entscheidung hat aber enorme Tragweite und wirft viele Fragen auf: Kann die Anmeldung über einen Dienst der Schulen erfolgen? Was für Auswirkugen hat das auf interessierte Schülerinnen ohne eine Schule im Hintergrund? Sollen die Schüler unterschiedlicher Schulen miteinander interagieren können? Wer haftet bei Verstößen gegen das Urheberrecht? Wer bezahlt den Betrieb der notwendigen Server? All diese (und vermutlich noch weitere) Fragen werden im Falle einer Umstellung zu klären sein.

A. Projektbeispiele

Dieses Kapitel beschreibt Ideen für Projekte, die sich gut mit BlattWerkzeug umsetzen lassen. Diese Auflistung ist dabei nicht vollständig, sie umfasst vielmehr jene Projekte, die gewissermaßen als „Proof-of-Concept“ im Rahmen der Entwicklung entstanden sind und sich auch für Lernende eignen könnten.

A.1. Interaktive Geschichten

Im englischsprachigen Raum hat sich für diese Art von Erzählung der Terminus „Choose Your Own Adventure“ durchgesetzt. In Deutschland existiert kein feststehender Begriff, stattdessen werden häufig exemplarische Buchreihen wie „Insel der tausend Gefahren“ stellvertretend für das Genre herangezogen. Für all jene, die auch mit diesen Begriffen nichts anfangen können, illustriert Abbildung 36, wie eine solche Geschichte funktioniert. Alternativ kann die Geschichte auch interaktiv unter cyoa.blattzeug.de ausprobiert oder in der deutschsprachigen Wikipedia nachgelesen werden[48], von dort wurde sie entnommen.

Anfang

Du bist im Kontrollraum eingesperrt. Wie kommst du hier nur wieder raus? Vor dir befindet sich ein großer roter Knopf und an der Wand darüber ist ein Schild angebracht. In die Nordwand ist eine schwere Stahltür eingelassen.

- Alles bleibt wie es ist! Du bleibst einfach stehen.
- Du drückst den roten Knopf!
- Türen sind zum Öffnen da.
- Schilder sind spannend! Mal schauen, was darauf steht?

Abbildung 36: Auswahlmöglichkeiten bei einer interaktiven Geschichte

Das Datenmodell (siehe Abbildung 37) für diese Geschichten besteht aus zwei Tabellen: `chapter` enthält die eigentliche Geschichte, also die Überschrift und den Rumpf aus Abbildung 36. Die Entscheidungsmöglichkeiten werden in `next_chapter` hinterlegt. Diese Tabelle stellt eine `m:n`-Beziehung zwischen Kapiteln her und reichert diese mit einem Entscheidungstext an.

[48] `https://de.wikipedia.org/wiki/Spielbuch#Beispiel`

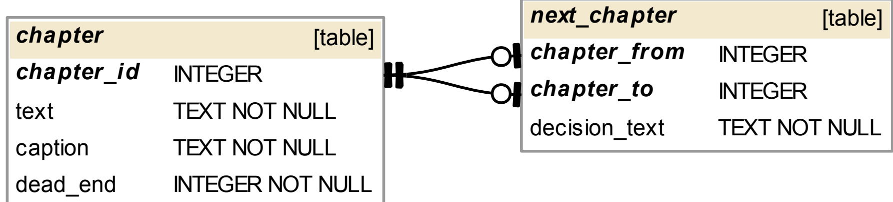


Abbildung 37: Schema für interaktive Geschichten

Die in Abbildung 36 zu sehende Kapitelseite benötigt drei Datenquellen: Per `GET`-Parameter wird das anzuzeigende Kapitel definiert. Die Abfragen `kapitel_einzeln` und `kapitel_optionen` benötigen eben jene Kapitel-ID als Parameter und liefern dann den Text des Kapitels selbst, sowie die ausgehenden Verweise auf andere Kapitel. Zur Darstellung der Optionen muss aktuell noch auf eine `HTML`-Einbettung zurückgegriffen werden, Abbildung 38 zeigt den Quelltext innerhalb von BlattWerkzeug.

Abbildung 38: Quelltext für die Anzeige eines Kapitels

A.2. Einfacher Blog mit Kommentaren

Dieses Beispiel erlaubt die Veröffentlichung von Artikeln durch einen Autor und gibt jedem Leser die Möglichkeit, einen Kommentar zu hinterlassen (Abbildung 39). Es lässt sich unter blog.blattzeug.de interaktiv ausprobieren.

Das Datenmodell speichert die eigentlichen Blogbeiträge in der Tabelle `article`, die dazugehörigen Kommentare in der Tabelle `comment`. Letztere Tabelle kann von einem nor-

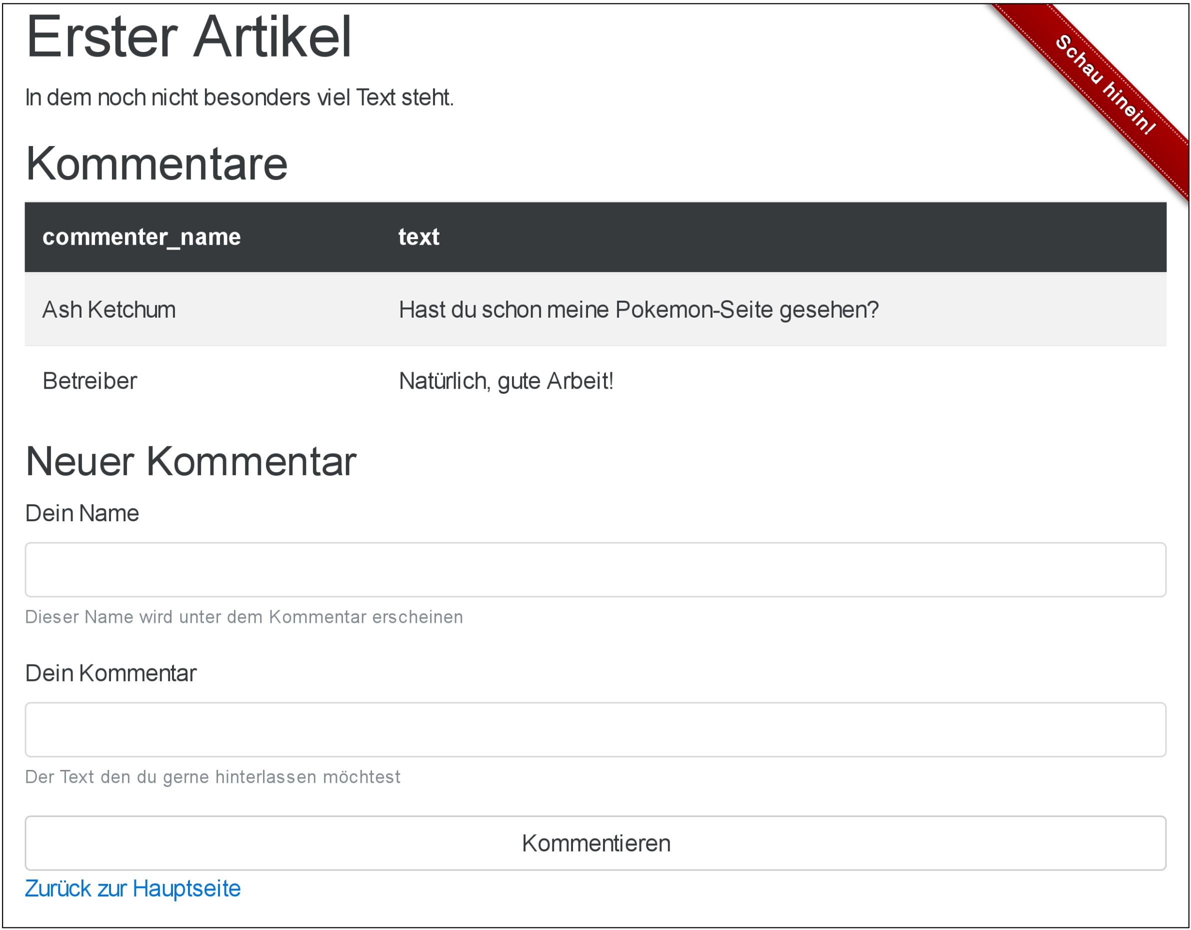


Abbildung 39: Ein (sehr kurzer) Blog-Artikel mit zwei Kommentaren

malen Endbenutzer über die Oberfläche mit weiteren Inhalten befüllt werden. Neue Artikel können hingegen nur von einem Entwickler angelegt werden. Da in BlattWerkzeug zum jetzigen Zeitpunkt allerdings noch keine Funktionen unterstützt werden, ist es nicht möglich die aktuelle Uhrzeit mit in den Datensatz aufzunehmen.

Die Seite zur Darstellung eines Blog-Artikels kommt mit den von BlattWerkzeug zur Verfügung gestellten Bedienelementen aus. Der anzuzeigende Artikel wird über einen `GET`-Parameter bestimmt, welcher als Eingabe für die Abfragen `artikel_detail` und `artikel_kommentare` verwendet wird. Um einen neuen Kommentar zu schreiben, muss der Endbenutzer in dem Formular seinen Namen und eine Nachricht hinterlassen. Der Knopf „Kommentieren“ ist mit einer mutierenden Abfrage verknüpft, welche diese Daten dann in der Datenbank hinterlegt.

Anders als bei der im vorigen Kapitel A.1 „Interaktive Geschichten“ vorgestellten Seite,

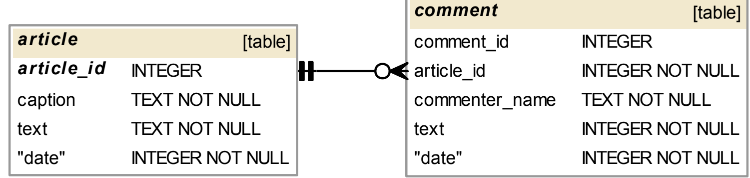


Abbildung 40: Schema für einen sehr einfachen Blog

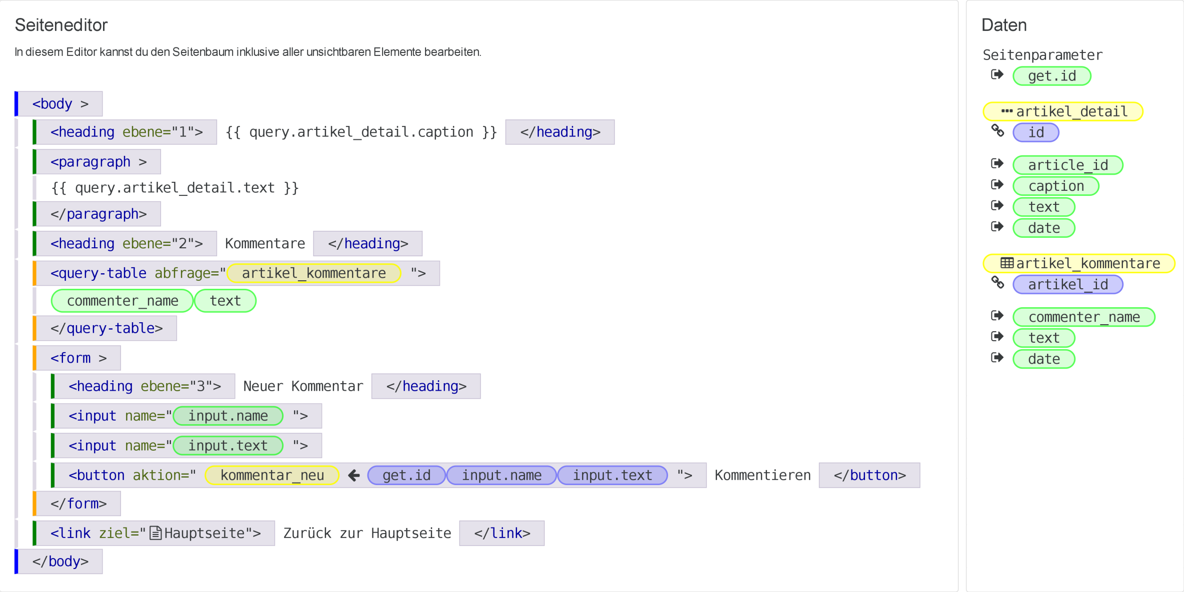
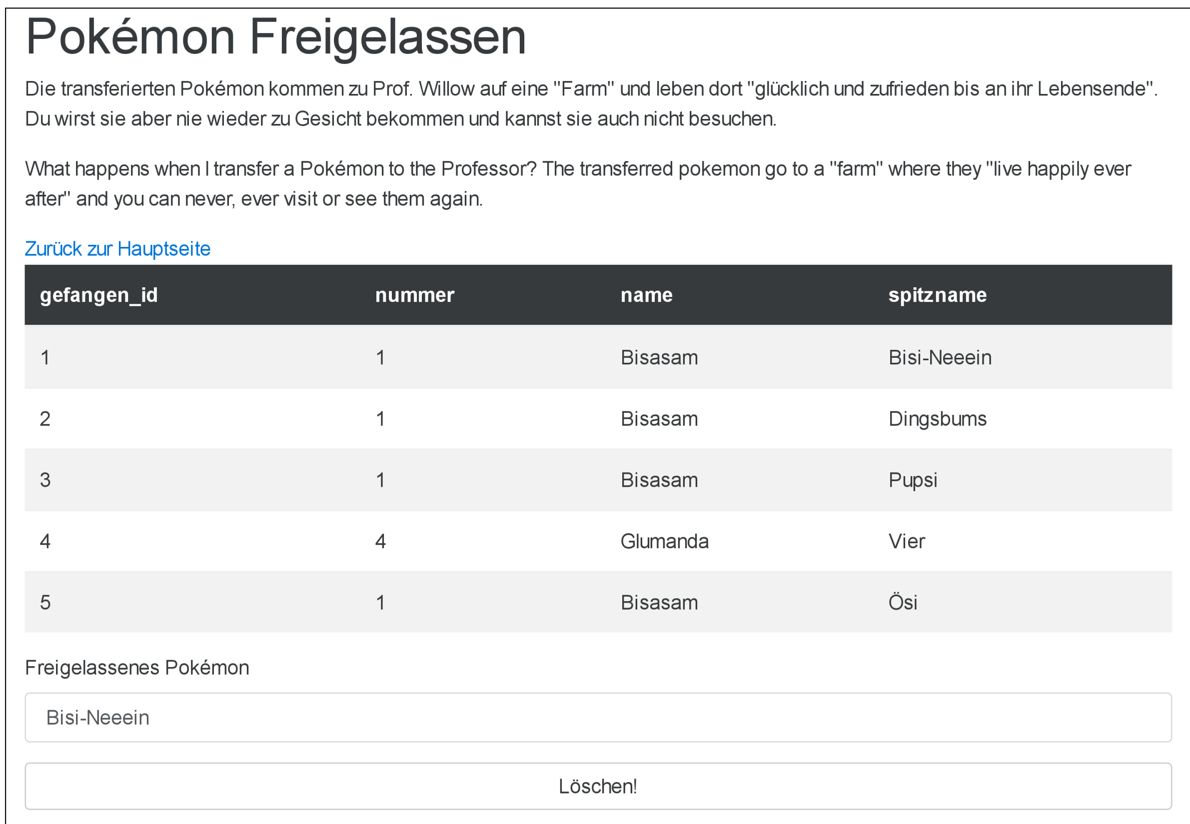


Abbildung 41: Quelltext für die Anzeige eines Blog-Artikels

ist für den Blog kein Einsatz von eingebettetem Quelltext notwendig. Für die Anzeige der Kommentare (siehe Abbildung 41) wird die eingebaute Datentabelle herangezogen, allerdings nur mit einer Teilmenge der verfügbaren Spalten. Schließlich spielt die `ID` des Artikels oder Kommentars in der Darstellung für Endanwender keine Rolle.

A.3. Pokémon Go

Dieses Projekt fungiert als eine virtuelle Vitrine, um die eigenen Spielfiguren aus Pokémon Go auszustellen. Der hinter dem Spiel stehende Datenbestand eignet sich prinzipiell


Pokémon Freigelassen

Die transferierten Pokémon kommen zu Prof. Willow auf eine "Farm" und leben dort "glücklich und zufrieden bis an ihr Lebensende". Du wirst sie aber nie wieder zu Gesicht bekommen und kannst sie auch nicht besuchen.

What happens when I transfer a Pokémon to the Professor? The transferred pokemon go to a "farm" where they "live happily ever after" and you can never, ever visit or see them again.

Zurück zur Hauptseite

gefangen_id	nummer	name	spitzname
1	1	Bisasam	Bisi-Neeein
2	1	Bisasam	Dingsbums
3	1	Bisasam	Pupsi
4	4	Glumanda	Vier
5	1	Bisasam	Ösi

Freigelassenes Pokémon

Bisi-Neeein

Löschen!

Abbildung 42: Löschen eines gefangenen Pokémon

gut zur Erläuterung von Problemen bei redundanten Daten. Der aktuell implementierte Stand gibt diesen Schwerpunkt aber noch nicht sehr gut wieder. Die ausgestellten Pokémon lassen sich von Endbenutzern sowohl anlegen als auch löschen (siehe Abbildung 42), was unter pokemongo.blattzeug.de ausprobiert werden kann.

Das aktuell implementierte Schema des Beispiels besteht aus zwei Tabellen (Abbildung 43). Die Bezeichnung „Pokémon“ wurde dabei vermieden, da sich eine komplette Spielfigur nur aus mehreren Tabellen zusammenstellen lässt. Die Tabelle `pokedex` enthält dabei gewissermaßen die Stammdaten der Spielfigur: Dabei handelt es sich um den übergeordneten Namen und einen Typ. Da ein Spieler jedes Pokémon auch mehrfach besitzen kann, sind diese Daten als unveränderlich anzusehen. Die konkreten Pokémon eines Spielers ergeben sich dann aus der Tabelle `gefangen`. Hier sind die Variablen Werte einer Spielfigur, also der frei wählbare Spitzname und die im Spiel verfügbare Stärke, hinterlegt.

An Abbildung 42 lässt sich erkennen, mit welcher Art Daten diese Tabellen befüllt wurden: Der Inhaber der Seite hat mehrere Varianten des Pokémon „Bisasam“ gefangen,

diese haben unterschiedliche Spitznamen bekommen und sind unterschiedlich stark.

Inhaltlich weist dieses Schema noch einige Verletzungen von gutem Datenbankdesign auf: Der Typ eines Pokémon ist nicht wahlfrei, sondern eine feste Aufzählung aus Werten wie „Feuer“, „Wasser“, ... Ferner kann jedes Pokémon inhaltlich nicht nur einen, sondern zwei Typen haben. An dieser Stelle müsste das Datenmodell also noch angepasst werden.

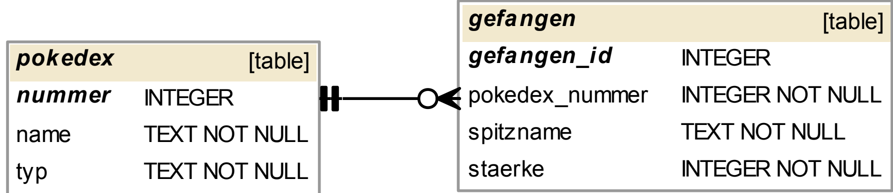


Abbildung 43: Schema für Pokémon Go

Aus Sicht eines Entwicklers ist an dem „Pokémon Freilassen“-Bilschirm vor allem interessant, dass die Abfrage zur Auflistung aller eigenen Pokémon (`Meine_Pokemon`, Abbildung 44) von zwei Bedienelementen verwendet wird (Abbildung 45). Sowohl die Datentabelle zur Anzeige der eigenen Fänge, als auch die Auswahl des zu löschenden Datensatzes in dem Formular machen davon Gebrauch.

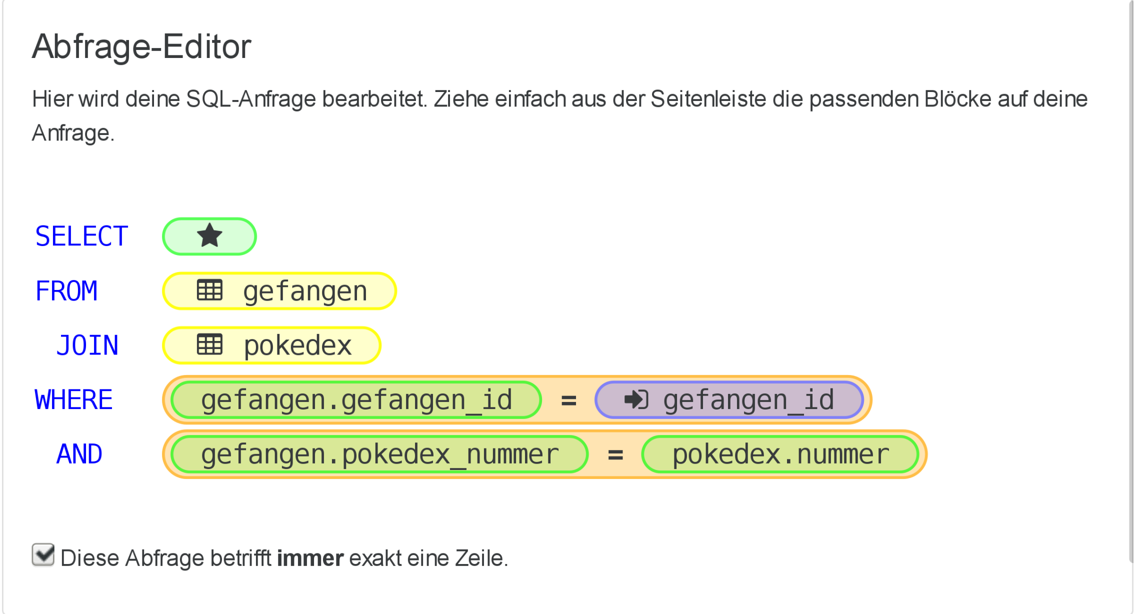


Abbildung 44: Abfrage zur Anzeige eines einzelnen Pokémon

Abbildung 45: Quelltext der Seite zum Löschen eines gefangenen Pokémon

B. Der Name BlattWerkzeug

Das aktuell BlattWerkzeug genannte Programm wurde im Verlauf der Entwicklung mehrfach umbenannt: Erst lautete der Arbeitstitel „Scratch for SQL“, dann „esqulino“ und nun „BlattWerkzeug“. Letzterer Name entstand dabei als deutsche Variante eines möglichen englischen Titels, nämlich „Pagetool“.

Namen für die im Rahmen dieser Thesis erstellten Software müssen sich zunächst den folgenden Kriterien stellen:

1. Wenn man den Namen nur ausgesprochen hört, soll sofort eindeutig sein, welchen Begriff man in die Suchmaschine seiner Wahl eingeben soll.
2. Er sollte auf zumindest einen der beiden Anwendungszwecke hinweisen, also entweder auf Oberflächen mit `HTML` oder Datenbanken mit `SQL`. Optimalerweise verbindet der Name natürlich beide Einsatzzwecke.

Leider erfüllen Weder „esqulino“ noch „BlattWerkzeug“ beide Kriterien so richtig zufriedenstellend.

Bei „esqulino“ hängt man sehr stark an der `SQL`-Abkürzung, sofern man die nicht kennt fällt die Wiedererkennung schwer. Gleiches gilt für die vielen möglichen Schreibweisen wie „SQLino“ oder „esqlino“. Und abgesehen davon ist ja auch die Sprechweise „Sequel“ für `SQL` nicht ungebräuchlich.

Die Verbindung zu Internetseiten wird bei „BlattWerkzeug“ in der Regel nur nach einer Erläuterung klar. Der gedankliche Weg von „Page“ zu „Seite“ und dann schließlich „Blatt“ ist möglicherweise doch ein wenig weit hergeholt ...

Alles in allem ist der Name daher noch nicht notwendigerweise endgültig. Bessere Alternativen werden dankend entgegengenommen.

C. Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Wedel, 31. Oktober 2016 Marcus Riemer

D. Literaturverzeichnis

[1] Robert Grimm und Oliver Scholle. *Informatik 2*. ger. Hrsg. von Thomas Kempe und Annika Löhr. Druck A, 1. Schöningh-Schulbuch. OCLC: 942763242. Paderborn: Schöningh, 2015. ISBN: 978-3-14-037127-8.

[2] Peter Hubwieser und Mattias Spohrer, Hrsg. *Tabellenkalkulationssysteme, Datenbanken*. ger. 1. Aufl., 5. Dr. Informatik Lehrwerk für Gymnasien ; 2, [Schülerbd,] OCLC: 934924890. Stuttgart: Klett, 2011. ISBN: 978-3-12-731668-1.

[3] Herbert Klaeren und Micheal Sperber. *Die Macht Der Abstraktion: Einführung in Die Programmierung*. Vieweg+Teubner Verlag, 2007. ISBN: 3-8351-0155-2.

[4] Land Schleswig-Holstein. *Lehrplan Für Angewandte Informatik in Der Sekundarstufe I*. 2010.

[5] Land Schleswig-Holstein. *Lehrplan Für Angewandte Informatik in Der Sekundarstufe II*. 2002.

E. CD-ROM

Alle auf der beigefügten CD-ROM enthaltenen Daten sind auch im BlattWerkzeug-Git-Repository unter `http://blattwerkzeug.de/forward/git-repository` verfügbar.