Bachelorthesis

Konzeption und Implementierung eines React-Frameworks zur flexiblen Erweiterung digitaler Datenbestände

Prüfer(in):
Prof. Dr. Christian Soltenborn

Verfasser(in):
Thomas Benjamin Hopf
101485

Heinrichstrasse 23
40764 Langenfeld

Wirtschaftsinformatik
Cyber Security

Eingereicht am:
10.06.2025

Sperrvermerk

Diese Arbeit enthält vertrauliche Informationen über die Firma Unternehmen GmbH. Die Weitergabe des Inhalts dieser Arbeit (auch in Auszügen) ist untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch nicht in digitaler Form - angefertigt werden. Auch darf diese Arbeit nicht veröffentlicht werden und ist ausschließlich den Prüfern, Mitarbeitern der Verwaltung und Mitgliedern des Prüfungsausschusses sowie auf Nachfrage einer Evaluierungskommission zugänglich zu machen. Personen, die Einsicht in diese Arbeit erhalten, verpflichten sich, über die Inhalte dieser Arbeit und all ihren Anhängen keine Informationen, die die Firma Unternehmen GmbH betreffen, gegenüber Dritten preiszugeben. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma Unternehmen GmbH und des Verfassers.

Die Arbeit oder Teile davon dürfen von der FHDW einer Plagiatsprüfung durch einen Plagiatsoftware-Anbieter unterzogen werden. Der Sperrvermerk ist somit im Fall einer Plagiatsprüfung nicht wirksam.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1 Einleitung

Relationale Datenbanken sind weiterhin die Standardlösung für die Speicherung und Verarbeitung betrieblicher Daten in Unternehmen[1]. Durch ihre rigide Struktur ermöglichen sie eine klare Trennung zwischen Datenstruktur und Anwendungsebene, wodurch eine hohe Datenintegrität sichergestellt wird[2];[3]. Diese rigide Struktur kann jedoch auch zu Einschränkungen führen, insbesondere wenn Tabellen zur Laufzeit erweitert werden müssten, um eine flexible und nutzerspezifische Erweiterung zu ermöglichen[4].

Auch im betrieblichen Ablauf der Bayer AG sind relationale Datenbanken mitsamt ihrer Einschränkungen allgegenwärtig. Dies gilt insbesondere für Abteilungen, die für Research and Development (R&D) zuständig sind. Alle Projekte, die im Rahmen von Forschungsaktivitäten durchgeführt werden, werden mitsamt zusätzlicher Attribute in einer Datenbank (Newport) hinterlegt, wodurch Budgetierung und Projekt-Planung effizient und effektiv möglich ist. Zwar besteht die Möglichkeit, dort viele Informationen zu hinterlegen, jedoch lassen sich die vorgegebenen Spalten nicht erweitern. Dadurch ist es nicht möglich, benutzerspezifische Informationen, die über den Horizont von Newport hinausgehen, zu speichern und weiterzuverarbeiten. Um dieses Problem zu lösen, soll eine Benutzeroberfläche entwickelt werden, die diese Speicherung ermöglicht.

Im Rahmen dieser Bachelor-Arbeit soll ein flexibles technisches Framework konzipiert und umgesetzt werden, mit welchem benutzerspezifische Attribut-Erweiterungen ermöglicht werden, ohne die bestehende Datenbankstruktur der Newport-Datenbank zu verändern. Dabei soll untersucht werden, wie sich die eingeführte Flexibilität mit den Anforderungen an Datenintegrität und Benutzerfreundlichkeit vereinbaren lassen.

Die Arbeit gliedert sich in folgende Kapitel: Zuerst sollen in Kapitel 2 wichtige grundlegende Konzepte und Begriffe rund um relationale Datenbanken und flexible Datenmodellierung erläutert werden. Daraufhin werden die Anforderungen an das zu entwickelnde

[1] Prasad, U. (2025).
[2] Codd, E. F. (1970).
[3] Date, C. J. (2003).
[4] Nadkarni, P. M. u. a. (2001).

Framework sowie die Zielsetzung für das Projekt definiert. Die folgenden Kapitel widmen sich der technischen Konzeption und Implementierung des Front- und Backends sowie einer abschließenden Evaluation der Ergebnisse. Abschließend folgt ein Ausblick auf mögliche zukünftige Entwicklungen.

2 Grundlagen und Hintergrund

Für die Umsetzung des Projekts ist theoretisches Wissen rund um Datenspeicherung -verarbeitung und -verteilung sowie Webdesign notwendig. Während grundlegendes Informatisches Fachwissen vorausgesetzt wird, werden alle weiteren für das Verständnis dieser Arbeit relevanten Konzepte im Folgenden erläutert.

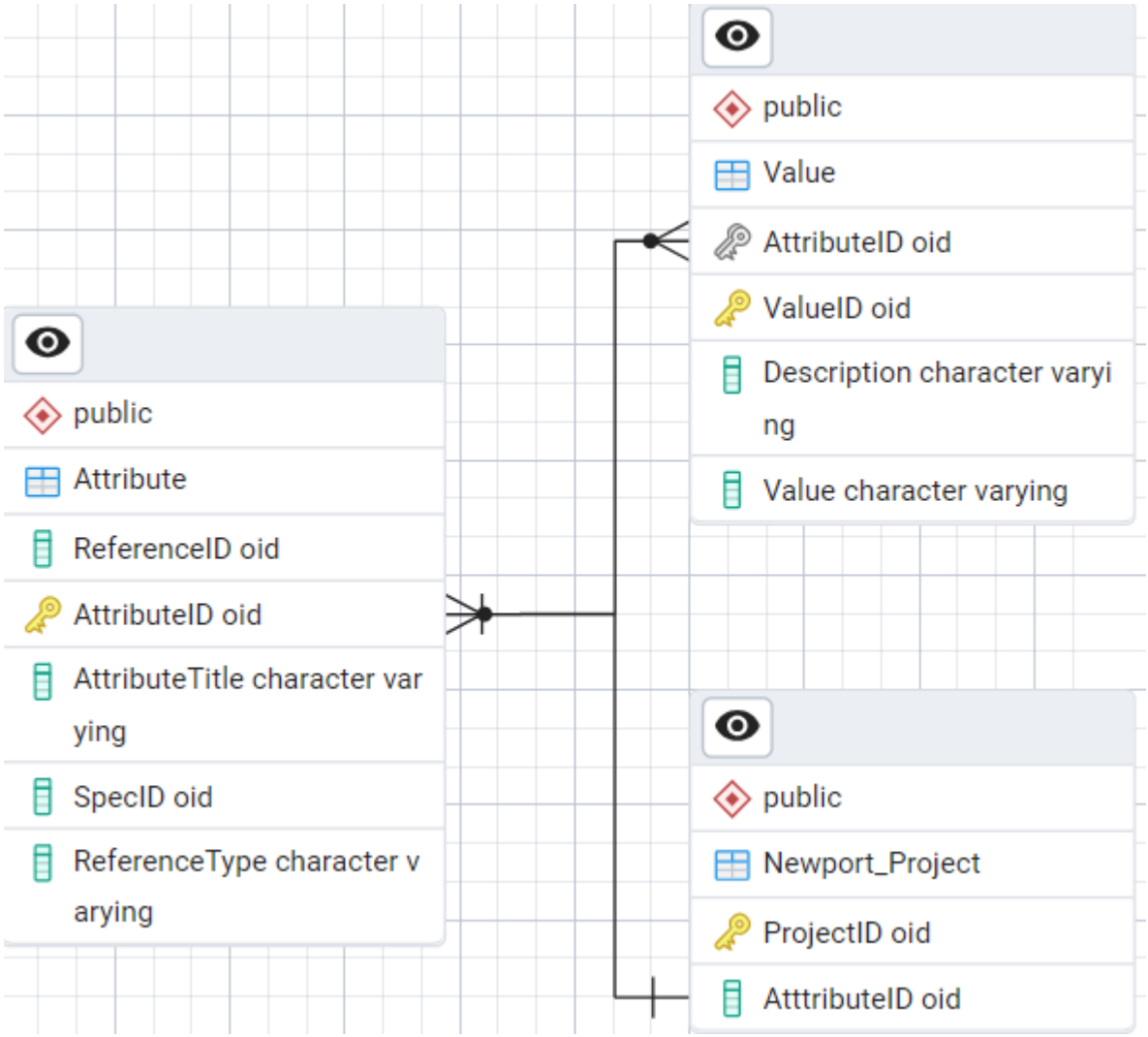
2.1 Fixed-Schema Datenbanken

Die Newport-Datenbank hat aufgrund ihrer großen Datenmenge und Nutzerbasis ein unveränderbares Schema. Das bedeutet, dass die vorgegebenen Spalten nicht bearbeitet, und auch keine neuen Spalten hinzugefügt werden können. Dadurch wird verhindert, dass die Anzahl an Spalten der Datenbank unkontrolliert wächst, was sich negativ auf Recheneffizienz und Übersichtlichkeit auswirken könnte. [5] Ein Nachteil dieses Konzepts ist jedoch die geringere Flexibilität. Ohne die Möglichkeit, neue Spalten hinzuzufügen, sind Nutzer auf die zum Zeitpunkt der Erstellung der Datenbank vorgesehenen Informationen beschränkt. Im Rahmen dieses Projekt soll diese Flexibilität durch Verwendung des Ättribute-Value-Pair=Modells in einer umschließenden Architektur gewährleistet werden, ohne aber die Vorteile einer Fixed-Schema Datenbank zu mindern

2.2 Attribute-Value-Pair Modell

Im Rahmen des Attribute-Value-Pair Modells (In Zukunft AVPM) werden Attribute, durch welche ein Objekt genauer beschrieben wird, in eine eigenständige Tabelle ausgegliedert. Diese Tabelle ist über einen Fremdschlüssel mit dem jeweils tragenden Objekt verbunden. Die Attribute sind in der Tabelle als Schlüssel-Wert-Paare gespeichert. Diese Struktur ermöglicht eine flexible Erweiterung der Attribute, ohne dass die Struktur der Datenbank verändert werden muss:

[5] quelle bitte



2.3 Technologischer Rahmen

Um dieses Modell im Backend umzusetzen, und dann im Front-End nutzbar zu machen, bedarf es verschiedener Technologien. Im Backend werden die Daten in einer Aurora PostgreSQL Datenbank gespeichert[6]. Aurora PostgreSQL (Im Folgenden Aurora) ist ein Dienst, der die Erstellung von relationalen Datenbanken ermöglicht. Auf dieser relationalen Datenbank ermöglicht eine in NodeJS [7] erstellte API den Zugriff auf und die Bearbeitung der gespeicherten Daten. Auf diese API greift dann widerum eine React-App zu, die dann ermöglicht, durch eine Benutzeroberfläche die API zur Kommunikation

[6]Diese Wahl beruht auf Standards im Unternehmen. Dazu mehr in der Evaluierung
[7]Bessere Kompatibilität mit React als alternativen

mit der Datenbank zu benutzen. Die Architektur ist bewusst mehrgliedrich gehalten, um Erweiterungen wie Daten-Redundanz zu ermöglichen [8] Die Wahl dieses technologischen Rahmens beruht auf vorhandenem Wissen und Infrastruktur im Unternehmen, sowie auch Abwägung der Entwicklungsgeschwindigkeit. Grundsätzlich stehen Enwticklern in der Web-Entwicklung jedoch viele Möglichkeiten zur Verfügung

2.4 Alternative Technologien

Für die Datenbank wäre es denkbar gewesen, anstatt einer relationalen auf eine sogenannte NoSQL Datenbanken zurückzugreifen. Eine solche Lösung wird auch im AWS-System angeboten. Der Grund, warum sich gegen diese Option entschieden wurde, liegt in der Natur der Datengrundlage. NoSQL eignet sich vor allem für große Mengen unstrukturierter Daten, während traditionelle Datenbanken besser mit strukturierten Daten umgehen können [9] Im Rahmen der Anwendung muss zwingendermaßen eine strenge Typisierung von Daten gegeben sein, da auf dessen Basis unter anderem Grafiken erstellt werden sollen. Aus diesem Grund ist eine strukturierte Datenbank für den speziellen Zweck des Projekts besser geeignet.

Die Entscheidung, NodeJS für die API zu verwenden, beruht hauptsächlich auf der zeitlichen Komponente der Entwicklung. React basiert auf einem NodeJS-Environment, weshalb es sich anbietet, die gleiche Umgebung für die APi zu verwenden. Grundsätzlich hätte auch jede andere Umgebung zur API-Erstellung ihren Zweck erfüllt, und diese Entscheidung beruht eher auf Bequemlichkeit. React hingegen als Umgebung wurde mit dem Hintergedanken der Modularität gewählt. Da das Projekt nicht nur eine einzene Anwendung, sondern eher einen Komplex kleinerer Tools beherbergen soll, bietet sich ein modulares Framework wie React an, wodurch dieses schon zu Beginn als Anforderung feststand, und als Basis aller weiteren Entscheidungen verwendet wurde. Zudem wird React unternehmensintern immer häufiger für verschiedenste Anwendungen verwendet, weshalb die Infrastruktur für das Deployment bereits aufgebaut ist.

[8] Unter anderem sollen die Daten in ein großes Daten Warehouse der Bayer Cropscience kopiert werden
[9] nosql source

3 Anforderungen und Abgrenzung

Die Anforderungen dieses Projekts gehen auf konkrete Bedürfnisse einer Nutzergruppe zurück, die im Arbeitsalltag intensiv mit dem System „Newport“ arbeitet. Aus diesen Bedürfnissen ergibt sich das zentrale Ziel dieser Arbeit.

3.1 Zielsetzung

Im Rahmen dieser Bachelor-Arbeit soll das Minimum Viable Product (MVP) des Projekts erarbeitet werden, also eine erste lauffähige Version mit den Kernfunktionen, die von der Nutzergruppe als unverzichtbar herausgearbeitet wurden. Entsprechend liegt in der Anforderungskonzeption der Fokus auf den sogenannten „Must-Have“-Anforderungen, ohne welche die Anwendung ihren Zweck verfehlen würde.

Die durch interne Gespräche erfassten Nutzeranforderungen lassen sich in zwei Kategorien einteilen. An erster Stelle stehen die funktionalen Anforderungen, die sich direkt auf die Interaktion mit der Anwendung beziehen. Ebenfalls wichtig sind jedoch auch nicht-funktionale Anforderungen, die sich auf Qualität, Sicherheit, Performance und Wartbarkeit der Lösung beziehen.

3.2 Funktionale Anforderungen

Im Rahmen des Minimum Viable Products (MVP) wurden folgende funktionale Anforderungen als unverzichtbar identifiziert. Sie bilden die Grundlage für die Nutzung des Systems durch Fachanwendende ohne tiefgehendes technisches Vorwissen.

- **Anlegen neuer dynamischer Attribute:** Nutzer*innen sollen die Möglichkeit haben, eigene Attribute anzulegen, die einem bestehenden Projektelement (z. B. einer Formulierung oder einem Projekt) zugewiesen werden können. Hierbei müssen folgende Angaben möglich sein:[10]
- **Zuweisung von Attributen zu Datenobjekten:** Bereits definierte Attribute sollen spezifischen Datenbankeinträgen zugeordnet und durch nutzerspezifische Werte ergänzt werden. Dies umfasst:

[10] Nadkarni, P. M. u. a. (2001).

Angabe	Pflichtfeld
Name des Attributs	Ja
Beschreibung	Nein
Datentyp (z. B. Text, Zahl, Datum)	Ja

Tabelle 1: Erforderliche Angaben beim Anlegen eines neuen Attributs

  - Auswahl eines vorhandenen Attributs
  - Eingabe eines entsprechenden Werts
  - Validierung des Werts entsprechend des Datentyps

[11]

- **Bearbeiten und Anzeigen von Attribut-Werten:** Im Benutzerinterface sollen alle bereits zugewiesenen Attribute mitsamt ihren Werten für ein bestimmtes Projektelement angezeigt und editierbar sein. Die Darstellung soll dynamisch erfolgen und sich an der Anzahl und Art der zugewiesenen Attribute orientieren.[12]
- **Löschen von Attribut-Zuweisungen:** Nutzer*innen sollen die Möglichkeit haben, bestehende Attribut-Wert-Paare wieder zu entfernen, ohne dabei das globale Attribut selbst zu löschen. Dadurch bleibt das Attribut für andere Einträge verfügbar.[13]
- **Benutzerführung und Validierung:** Die Oberfläche muss durch verständliche Beschriftungen, Platzhaltertexte, Tooltips oder andere visuelle Hinweise den Nutzer bei der Eingabe unterstützen. Validierungsfehler sollen unmittelbar und verständlich angezeigt werden.[14]

3.3 Nicht-funktionale Anforderungen

Neben den funktionalen Anforderungen, die die direkten Systemfähigkeiten betreffen, wurden verschiedene nicht-funktionale Anforderungen identifiziert. Diese beschreiben die Qualitätsmerkmale des Systems und sind essenziell für den erfolgreichen Betrieb in einem unternehmenskritischen Umfeld wie dem von Bayer.

[11] Vgl. Li, C. und Box, D. (2005).
[12] Vgl. Li, C. und Box, D. (2005).
[13] Vgl. Erdem, H. und Finin, T. (2009).
[14] Vgl. Nielsen, Jakob (2014).

- **Performance:** Das System muss eine hohe Reaktionsgeschwindigkeit aufweisen, um eine flüssige Nutzererfahrung zu gewährleisten. Die Ladezeit für die Anzeige eines Projektelements inklusive dynamischer Felder soll unter 500 ms liegen (bei bis zu 20 zusätzlichen Attributen). Schreiboperationen (z. B. Hinzufügen eines Attribut-Werts) sollen ohne spürbare Verzögerung erfolgen.[15]

- **Skalierbarkeit:** Die Lösung soll so ausgelegt sein, dass sie mit wachsender Datenmenge und Nutzerzahl ohne grundlegende Umstrukturierung betrieben werden kann. Die Datenbankstruktur muss große Mengen an dynamischen Attribut-Werten performant verarbeiten können. Die Architektur (Frontend, Backend, Datenbank) muss eine horizontale Skalierung ermöglichen (z. B. API-Lastverteilung).[16]

- **Datenintegrität:** Es muss sichergestellt werden, dass sämtliche gespeicherte Daten vollständig, korrekt und konsistent sind. Alle Eingaben durch die Nutzer:innen müssen auf Plausibilität und Formatkonformität geprüft werden. Die Datenbank muss durch Constraints und Foreign Keys inkonsistente Zustände verhindern.[17]

- **Wartbarkeit:** Die Codebasis soll modular und verständlich strukturiert sein, sodass zukünftige Weiterentwicklungen oder Fehlerbehebungen effizient durchgeführt werden können. Der Quellcode muss dokumentiert sein (Kommentare, Readmes, API-Spezifikationen). Wiederverwendbare Komponenten und Services sollen sauber gekapselt sein.[18]

- **Sicherheit:** Das System muss grundlegende Sicherheitsanforderungen erfüllen, um unberechtigte Zugriffe zu verhindern und sensible Daten zu schützen. Schreibende Aktionen (z. B. das Anlegen neuer Attribute) sollen nur autorisierten Nutzern erlaubt sein. Die API muss gegen typische Angriffe (z. B. SQL-Injection, CSRF) abgesichert sein.[19]

- **Benutzerfreundlichkeit (Usability):** Die Benutzeroberfläche muss so gestaltet sein, dass auch nicht-technische Anwender:innen effektiv mit dem System arbeiten können. Komplexe Prozesse wie das Anlegen neuer Felder sollen durch schrittweise

[15] Vgl. Galletta, D. F. u. a. (2006).
[16] Vgl. Leavitt, Nathan (2010).
[17] Vgl. Date, C. J. (2003).
[18] Vgl. Parnas, David L. (1972).
[19] Vgl. OWASP Foundation (2017).

geführte Dialoge unterstützt werden. Fehlerzustände (z. B. ungültige Eingaben) müssen klar und nachvollziehbar kommuniziert werden.[20]

3.4 Abgrenzung

Es ist zwingend notwendig, sich im Rahmen dieser Bachelorarbeit auf das MVP der Anwendung zu fokussieren, um eine abschließende Evaluation zu gewährleisten. Dementsprechend gibt es relevante Anforderungen an das fertige Produkt, die im Rahmen dieser Arbeit nicht umgesetzt werden. All diese Anforderungen sollen im Folgenden der Vollständigkeit halber aufgeführt und begründet werden.

- **Rollen- und Berechtigungssystem:** Viele verschiedene Nutzer werden für ihre jeweiligen Vorhaben in dieser Anwendung Attribute anlegen und diesen Werte zuweisen. Da künftig auch vertrauliche Daten gespeichert werden könnten, ist perspektivisch ein Berechtigungssystem erforderlich. Vorerst werden Nutzer jedoch angewiesen, keine solchen Daten zu speichern, bis dieses System fertig implementiert ist, was voraussichtlich erst nach Ende des Bearbeitungszeitraums dieser Arbeit geschehen wird.
- **Internationalisierung:** Das fertige Produkt soll, wie auch in vielen anderen Anwendungen der Fall (z. B. SAP), in vielen verschiedenen Sprachen verfügbar sein. Das soll erreicht werden, indem dynamische Language Files eingesetzt werden, die auch eigens von Nutzern bereitgestellt werden können. Eine solche Implementierung ist jedoch zeitaufwendig und aufgrund der Sprachkenntnisse der Nutzer nicht von elementarer Wichtigkeit und wird deshalb in dieser Arbeit keine Verwendung finden.
- **Integration in andere Systeme:** Das Projekt soll zwar publiziert werden, um Nutzer-Rückmeldungen erhalten zu können, jedoch soll es noch nicht in die restliche IT-Landschaft integriert werden, da das über lange Zeit und unter großer Vorsicht geschehen muss, um Kompatibilität zu garantieren.

Zusätzlich zu den hier aufgeführten Anforderungen wurden Erfolgskriterien definiert, anhand derer die Zielerreichung in Kapitel 4 überprüft wird.

[20] Vgl. Nielsen, Jakob (2016).

4 Systemdesign und Herausforderungen

Die Architektur ist im Sinne der Übersicht und Erweiterbarkeit klar in drei Teile aufgeteilt. Zur Datenspeicherung wird eine Aurora-Datenbank verwendet, mit welcher über eine NodeJS API von einem in React erstellten User Interface kommuniziert werden kann. Die Datenbank soll in der Lage sein, Attribut-Wert-Paare flexibel beherbergen zu können, und dabei die bestmögliche Performanz aufweisen.

4.1 Datenbankarchitektur

Um die Erweiterbarkeit der Anwendung zu garantieren, muss die Datenstruktur fest sein. [21] Trotzdem muss die Datenbank aber in der Lage sein, flexibel unterschiedliche Attribute und Werte speichern zu können. Das soll folgende Datenstruktur erreichen:

[21] quelle bitte

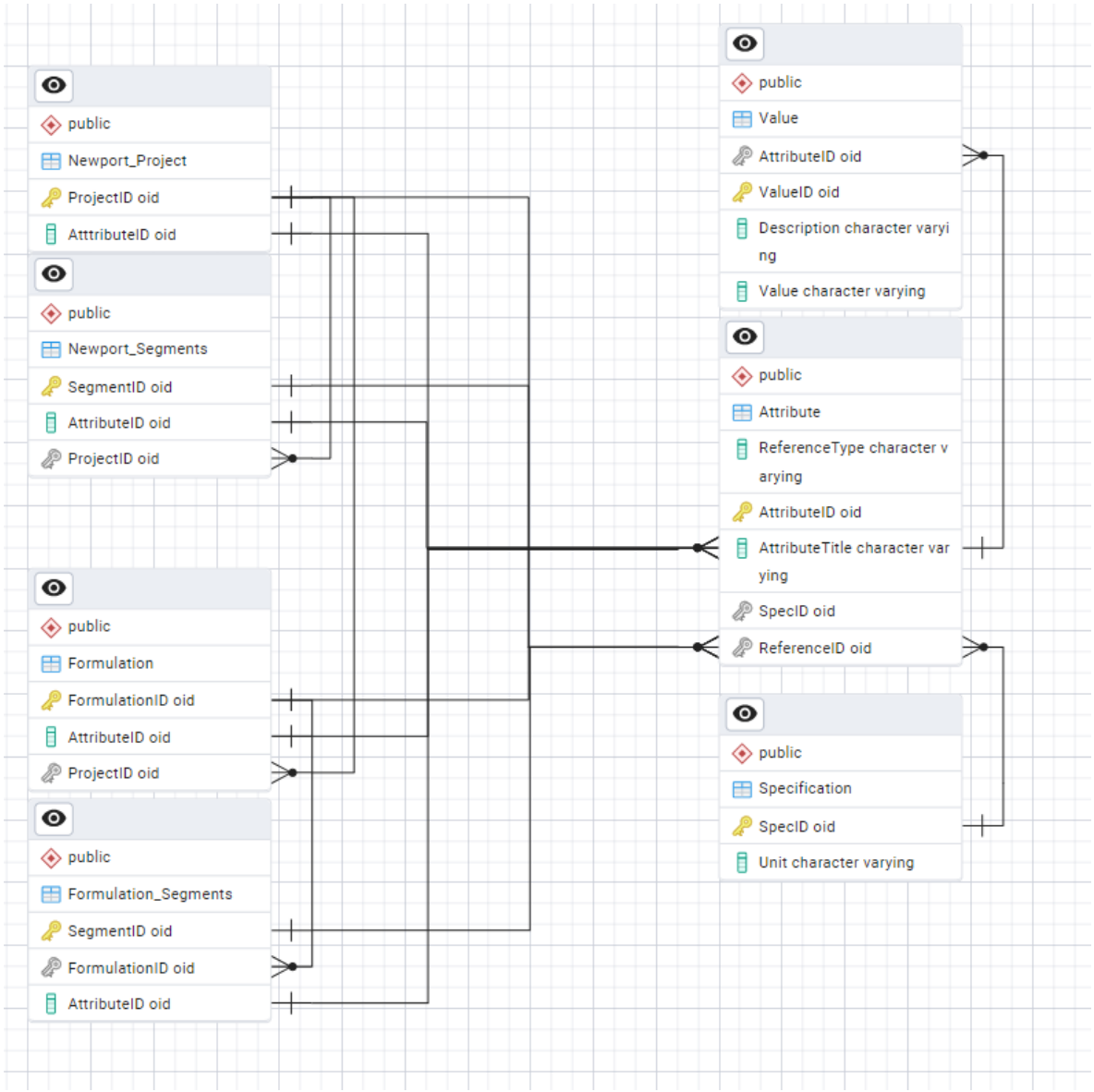


Abbildung 1: Modell der Datenbank

Die Datentruktur lässt sich in drei wesentliche Schichten aufteilen: An erster Stelle stehen die festen Datenobjekte, welche gleichlautend aus der bestehenden Infrastruktur übernommen werden. Diese Schicht bildet das Bindeglied zwischen der bereits existierenden Datenlandschaft und der von dieser Anwendung angehängten flexiblen Daten. Darauf folgt die dynamische Attributzuordnung, die über Fremdschlüssel direkt mit der ersten Schicht verbunden ist. Sie dient zur Verlinkung verschiedener Attribute mit ihren jeweiligen Eltern-Tabellen, speichert zusätzlich aber auch grundlegende Informationen über die Attribute ab. Die dritte Schicht beherbergt die Werte, und verbindet diese mit

ihren zugehörigen Attributen. In dieser Schicht sind Informationen zu der Natur der Werte, aber auch zum Inhalt abgespeichert. Im Folgenden sollen diese drei Schichten mit ihren jeweiligen Tabellen genauer beleuchtet werden.

4.1.1 1. Domänenspezifische Tabellen

„Newport_Project“ bildet das übergeordnete Forschungsprojekt und ihre zugehörigen Daten ab, wobei „Formulation“ nur ein Forschungsobjekt beschreibt, welches an ein Projekt geknüpft sein kann, aber nicht unbedingt muss:

Tabelle 2: Newport_Project

Spalte	Datentyp	Schlüssel?	Beschreibung
ProjectID	Object Identifier (oid)	Primärschlüssel	Primärer Identifier für den Eintrag
AttributeID	Object Identifier (oid)	Fremdschlüssel	Verweis auf zugewiesene Attribute für das jeweilige Projekt

Quelle: Eigene Darstellung

Tabelle 3: Formulation

Spalte	Datentyp	Schlüssel?	Beschreibung
FormulationID	Object Identifier (oid)	Primärschlüssel	Primärer Identifier für den Eintrag
AttributeID	Object Identifier (oid)	Fremdschlüssel	Verweis auf zugewiesene Attribute für das jeweilige Projekt
ProjectID	Object Identifier (oid)	Fremdschlüssel	Verweis auf ein gegebenenfalls zugewiesenes Newport-Projekt

Quelle: Eigene Darstellung

Tabelle 4: Newport_Segments

Spalte	Datentyp	Schlüssel?	Beschreibung
SegmentID	Object Identifier (oid)	Primärschlüssel	Primärer Identifier für den Eintrag
AttributeID	Object Identifier (oid)	Fremdschlüssel	Verweis auf zugewiesene Attribute für das jeweilige Segment
ProjectID	Object Identifier (oid)	Fremdschlüssel	Verweis auf das zugehörige Newport-Projekt

Quelle: Eigene Darstellung

Tabelle 5: Formulation_Segments

Spalte	Datentyp	Schlüssel?	Beschreibung
SegmentID	Object Identifier (oid)	Primärschlüssel	Primärer Identifier für den Eintrag
Formulation	Object Identifier (oid)	Fremdschlüssel	Verweis auf zugewiesene Attribute für das jeweilige Segment
ProjectID	Object Identifier (oid)	Fremdschlüssel	Verweis auf das zugehörige Newport-Projekt

Quelle: Eigene Darstellung

4.1.2 2. Attributebene

Tabelle 6: Attribute

Spalte	Datentyp	Schlüssel?	Beschreibung
ReferenceType	String (oid)	/	Automatisch gesetzte wörtliche Referenz auf den Owner des Attributs
ReferenceID	Object Identifier (oid)	Fremdschlüssel	Direkter Verweis auf den Owner des Attributs
AttributeID	Object Identifier (oid)	Primärschlüssel	Verweis auf das zugehörige Attribut
AttributeTitle	String	/	Titel des Attributs
SpecID	Object Identifier (oid)	Fremdschlüssel	Verweis auf die zugehörige Spezifikation

Quelle: Eigene Darstellung

4.1.3 3. Werteebene und Spezifikation

Tabelle 7: Value

Spalte	Datentyp	Schlüssel?	Beschreibung
AttributeID	Object Identifier (oid)	Fremdschlüssel	Verweis auf das zugehörige Attribut
ValueID	Object Identifier (oid)	Primärschlüssel	Primärer Identifier für den Eintrag
Description	String	/	Name des Werts
Value	String	/	Wert

Quelle: Eigene Darstellung

Tabelle 8: Specification

Spalte	Datentyp	Schlüssel?	Beschreibung
SpecID	Object Identifier (oid)	Primärschlüssel	Primärer Identifier für den Eintrag
Unit	String	/	Einheit

Quelle: Eigene Darstellung

4.2 Herausforderungen

Im Folgenden sollen die Herausforderungen, die die Anforderungen an das Projekt posieren, und mögliche Lösungsansätze für diese diskutiert werden.

4.2.1 Datenkonsistenz

Aufgrund der hohen Flexibilität der einzutragenen Daten muss die Datenbank modular aufgebaut sein. Um die Datenkonsistenz zu garantieren, muss die API (Direkte Manipulation ist in der Architektur nicht vorgesehen) sicherstellen, dass keine inkorrekten Daten gespeichert werden. Dafür müssen alle API-Requests noch vor Absendung auf Validität geprüft werden. Schlägt die Prüfung an, soll der Nutzer über den Fehler informiert werden. Da die Datenbank modular aufgebaut ist, muss die Erweitbarkeit und Performanz sichergestellt werden.

4.2.2 Referentielle Integrität

Die referentielle Integrität wird durch sogenannte „Constraints“ schon im Datenbank-Management-System (DBMS) sichergestellt. Dadurch, dass Foreign Keys als solche deklariert werden, überprüft das DBMS bei Erstellung eines neuen Eintrags mit einem Foreign Key, ob der referenzierte Eintrag auch tatsächlich existiert. Ist das nicht der Fall, wird die Erstellung des Eintrags verhindert.

4.2.3 Erweiterbarkeit und Performanz

Um die Erweiterbarkeit zu garantieren, speichert die Datenbank auch Informationen, die im aktuellen Funktionsumfang noch nicht zwingend Erforderlich sind, jedoch bei eventuell auftretenden Erweiterungen notwendig werden können. Ein Beispiel dafür ist die Tabelle „Specification“. Sie speichert genauere Informationen über die Eigenschaften der Werte wie der Einheit ab. Das ist noch nicht erforderlich, da alle vorhersehbaren Daten Numerisch sind. Sobald sich das aber ändert kann die Erweiterung ablaufen, ohne die Datenbank-Struktur zu verändern. Das ist wichtig, da für eine solche Veränderung das gesamte Backend bearbeitet werden muss. Es ist effizienter, die möglichen Veränderungen sofort einzuplanen. Dabei darf die Performanz aber nicht vernachlässigt werden. Durch die vielen Sicherheitsmechanismen, hinter welchen die Daten auf AWS gelagert werden, wird die Response-Time ohnehin verlängert, der restliche Prozess muss sich daran also anpassen. Der Nutzer soll jederzeit über den Fortschritt ausstehender Anfragen informiert werden, die Anwendung muss, auch während einer laufenden Anfrage, weiter bedienbar sein und alle Prozesse rund um den Datenabruf sollten möglichst effizient gestaltet sein. Ein Beispiel für eine optimierende Planung in der Architektur ist die Spalte „ReferenceType“. Durch diese Spalte, die automatisch gesetzt wird, müssen die großen Tabellen von Attribut und Attribut-Owner nicht miteinander verbunden werden, um sie zuzuordnen, wodurch (bei größeren Datenmengen in den Tabellen) die Response Time deutlich verbessert wird. Trotz dieser Optimierungen darf aber die Architektur nicht so komplex werden, dass die Wartbarkeit darunter leidet.

4.2.4 Wartbarkeit und Benutzerfreundlichkeit

Die Gesamtarchitektur muss stets, trotz aller Herausforderungen, simpel genug sein, um mit möglichst geringem zusätzlichen Arbeitsaufwand gepflegt werden zu können. Dementsprechend muss die Gesamtheit der Architektur bis ins Detail dokumentiert und alle Entscheidungen festgehalten werden. Neben den Entwicklern, die für die Wartung zuständig sind, müssen aber auch die Nutzer in der Lage sein, die Anwendung zu verstehen und effektiv anzuwenden. Deshalb muss das User Interface so selbsterlklärend wie möglich sein, und alle weiteren Informationen in einer User-Dokumentation festgehalten werden, auf welche klar und eindeutig verwiesen wird.

5 Datenmodellierung

Im Sinne der Modularität basiert die Datenmodellierung auf dem Entity-Value-Attribut-Konzept (EAV). Dieses Konzept ermöglicht die Erweiterung des starren relationalen Schema um eine variable Anzahl benutzerdefinierter Attribute, ohne dabei eine Schema-Veränderung zu erfordern. Dadurch sind die dynamischen Attribute von den Domänenobjekten (Projekte und Formulationen) getrennt. Im Folgenden soll der Aufbau und die Funktionsweise des Datenmodells genauer erläutert werden.

5.1 ER-Modell und Tabellenübersicht

Das relationale Schema gliedert sich in 3 Hauptblöcke: Domänentabellen, Attributdefinition und der Werteebene.

5.1.1 Domänentabellen

„Newport_Project“ verbindet die Datenbank über den Primärschlüssel „ProjectID“ mit der Newport-Datenbank aus dem CropScience Warehouse (CSW). Das ermöglicht indirekten Zugriff auf externe Projektdaten. „Newport_Segments“ beinhaltet sogenannte Segmente[22] für die jeweiligen Newport-Projekte, und (sofern gegeben) deren Attribute. „Formulation“ und „Formulation_Segments“ fungieren entsprechend für zu speichernde Formulationen.

5.1.2 Attributdefinition

„Attribute“ enthält alle potentiell verwendbaren, dynamischen Felder. Jede Zeile verfügt über eine eindeutige „AttributeID“, einen entsprechenden „AttributeTitle“, und sowohl eine direkte („ReferenceType“), als auch eine indirekte Referenz(„ReferenceID“) auf den Attribut-Owner. Mit der „SpecID“ wird zusätzlich auf eine Tabelle verwiesen, in der genauere Informationen über das Attribut (bspw. Einheit) hinterlegt werden können.

[22] Segmente sind in der Enwticklung einer Formulation beispielsweise verschiedene Länder, in denen das Produkt lizensiert werden soll

5.1.3 Werteebene

„Value“ speichert die benutzerdefinierten Werte, mit denen das Attribut angelegt wurde. Wesentliche Spalten sind „ValueID“, der Primärschlüssel, „AttributeID“, die Referenz auf das zugehörige Attribut, „Description“ für eine mögliche genauere Beschreibung und „Value“, worin der eigentliche Wert gespeichert wird.

5.2 Beziehungen und Referentialität

Ein Domänenobjekt kann über die AttributeID mit beliebig vielen Attributen verbunden werden. Ein Attribut kann widerum nur einem Domänenobjekt zugewiesen werden. Die referentielle Integrität wird durch die Fremdschlüssel-Paarungen sichergestellt, wonach kein Attribut ohne Verweis auf ein Domänenobjekt erstellt werden kann. Die API ist hingegen dafür zuständig, dass ein Attribut nicht auf mehrere verschiedene Domänenobjekte verweist, indem es benutzerdefiniertes Setzen der AttributeID verhindert.

5.3 Datenkonsistenz und Validierung

Das EAV-Modell sieht vor, dass die Datenintegrität durch die mit der Datenbank verbundene API sichergestellt wird, indem eingegebene Werte stets auf Validität geprüft werden. Folgende Prüfungen sind für die API vorgesehen:

5.3.1 Datentypprüfung

Jedes Attribut hat in der Specification einen hinterlegten Datentyp. Sollte der vom Benutzer angegebene Datentyp nicht in der Menge unterstützter Datentypen vorhanden sein, soll die Erstellung des Attributs verhindert werden und eine entsprechende Fehlermeldung ausgegeben.

5.3.2 Pflichtfelder und Standardwerte

Um eine reibungslose Behandlung fehlerhafter Nutzerangaben sicherzustellen, soll schon das Front-End Pflichtfelder als solche markieren, und das Abschicken der Form verhindern, bis diese ausgefüllt sind. Das Setzen der Schlüssel übernimmt jedoch das Backend, weshalb diese, obwohl sie Pflichtfelder sind, nicht vom Nutzer gesetzt werden müssen.

5.3.3 Transaktionen

Das Anlegen eines neuen Attributs und das direkte Zuweisen eines ersten Werts sollen in einer Transaktion gebündelt sein, um zu verhindern, dass Attribute ohen zugehörige Werte existieren.

5.4 Beispielhafte Speicherung

Angenommen, ein Nutzer legt für „Projekt A“ ein neues Attribut „Versuchsdauer“ (Einheit Integer) an und vergibt den Wert „3“, dann sähen die Abläufe und Datenbankeinträge folgendermaßen aus:

Tabelle 9: Attributdefinition

AttributeID	AttributeTitle	ReferenceType	SpecID
42	Versuchsdauer	Newport_Project	2

Tabelle 10: Specification (Einheitenzuordnung)

SpecID	Unit
2	Integer

Tabelle 11: Speicherung eines Attribut-Wert-Paares

ValueID	AttributeID	Description	Value
101	42	null	3

Durch diese Trennung bleiben das feste Schema in „Newport_Project“ und die dynamischen Erweiterungen klar voneinander getrennt, und trotzdem werden alle zusätzlichen relevanten Informationen abgespeichert.

Mit diesem Datenmodell können beliebig viele neue Attribute hinzugefügt werden, ohne das Grundschema der Domänentabelle anzupassen.

6 Bakend-Implementierung

Im Backend übernimmt eine mit Node.js implementierte API die Geschäftslogik. Da die Daten in AWS hinter einer Firewall liegen, sitzt die API in einer EC2-Instanz, die eine Schnittstelle zwischen der gesicherten AWS-Landschaft und externen Apps bildet. Zustäzlich wird API Gateway, ein AWS-Serice verwendet, um den automatisierten Zugriff auf die API zu ermöglichen. Gespeichert sind die Daten in einer Aurora PostgreSQL Datenbank.

6.1 Datenbank

Zur Implementierung der Datenbank auf der EC2-Instanz muss zuerst eine Verbindung mit derlokalen Datenbankmanagement-System aufgebaut werden. Dafür wird die AWS-CLI verwendet.[23] Mit ihr wird dann[24] ein Tunnel zwischen dem lokalen Port 2222[25] und dem TCP-Port 22 der EC2-Instanz geöffnet: Dieser Tunnel ist zwingend erforderlich, da

Listing 1: SSH-Tunnel mit AWS-CLI

```
aws --profile {Profile-ID} ec2-instance-connect open-tunnel --instance-id {Ins
```

aufgrund von Sicherheitsrichtlinien Bayer's die EC2-Instanz keine öffentliche IP haben darf.
In PgAdmin kann nun eine Verbindung zu der Datenbank hergestellt werden. Dafür werden folgende Informationen benötigt:

1. Host name/Adresse: Die URL der Datenbank auf AWS-Ebene (name.id.server.rds.amazonaws.c
2. Port: 5432 (Standard Postgres-Port)
3. Maintenaince Database: Name der Datenbank auf AWS
4. UserName: Name des Nutzers, mit dem auf die Datenbank zugegriffen werden soll
5. Parameter „SSL Mode“: „prefer“
6. Use SSH Tunneling: „enabled“
7. Tunnel Host: localhost (da der SSH-Tunnel auf dem localhost aufgesetzt ist)
8. Tunnel Port: 2222 (entspricht –local-port in der CLI)

[23] https://aws.amazon.com/cli/
[24] Nach Authentifizierung mit aws login –{user}
[25] Der Port ist frei wählbar, solange er nicht reserviert ist (bspw. 22 ist nicht wählbar, da belegt durch TCP)

9. Authentication: „Identity File“
10. Identity File: Hier das .pem-File hinterlegen[26]

Mit diesen Einstellungen kann der Server gespeichert werden, und ein Zugriff auf die Datenbank ist möglich. In PgAdmin kann nun wie üblich das geplante Schema umgesetzt werden. Nun müssen die Daten in React abrufbar gemacht werden.

6.2 API

Für die API muss zuerst eine Infrastruktur in AWS errichtet werden, die eine sichere Verbindung zwischen AWS-Externen Clients und den AWS-Internen Daten garantieren kann. Diese Infrastruktur sieht folgendermaßen aus: (Hier bild) An erster Stelle steht die Virtual Private Cloud (VPC). Auf ihr liegt die gesamte Infrastruktur. Inbound und Outbound traffic aller IP-Adressen sind vollständig blockiert, um die Sicherheit der Daten zu gewährleisten. Auf dieser VPC liegt die Datenbank, auf welche Zugriff durch eine EC2-Instanz ermöglicht wird. Die Datenbank ist eine Aurora Postgres Datenbank in Standardausführung[27]. Die EC2-Instanz entspricht ebenfalls der Standardausführung, und ist so eingerichtet, dass Inbound-traffic auf TCP-Ebene ausgehend von AWS-Internen IP-Adressen möglich ist. Zusätzlich ist die EC2-Instanz mit einem Network Load Balancer (NLB) ausgestattet, wodurch andere AWS-Programme direkt auf Elemente innerhalb der VPC zugreifen können. Dieser Zugriff ist durch eine Target Group möglich, über den ein NLB standardmäßig verfügt, und welcher in diesem Fall direkt auf das VPC zeigt. In einem NodeJS-Programm, welches direkt auf der EC2-Instanz liegt, wird eine Proxy errichtet, mit der eine Verbindung nach außen hergestellt wird: In diesem Code-Snippet wird die Proxy auf den Localhost gerichtet, in der Produktivumgebung zeigt die Proxy auf die URL der Umgebung. Die über die Proxy weitergeleiteten Requests werden von API Gateway verarbeitet, auf welchem die oben formulierten Methoden umgesetzt sind. Jede Methode entspricht einer Lambda-Funktion, über welche die Interaktion mit der Datenbank ermöglicht wird: Zuerst wird in Form der Kontextvariable „pool“ der Gesamtkontext (alle Informationen zu) der Datenbank gespeichert. Mit diesen Informationen wird dann eine Query auf der Datenbank ausgeführt, und das Ergebnis zurückgesendet. Die Lambda-Funktionen sind in API Gateway einer eigenen URL zugeordnet, die über

[26] Key-Files können im AWS-Dashboard unter EC2 erstellt werden
[27] Standardausführung festgelegt durch den Unternehmensinternen Softwarekatalog

die Proxy auch außerhalb von AWS bei gegebener Authentifizierung abrufbar ist. Diese wird durch Cognito abgewickelt.

6.3 Authentifizierung mit Cognito

In Cognito liegt, speziell für die Authentifizierung von Nutzern für dieses Projekt, ein User-Pool. Dieser hat eine eigene Domain, in der hinterlegte Nutzer ihre Credentials angeben können. Werden die Credentials bestätigt, stellt Cognito ein Auth-Token zur Verfügung, mit welchem dann der Zugriff auf die API möglich ist. Im Frontend wird mit einem Login-Button ein Link zu der Cognito-Auth-Seite hergestellt, der den User nach erfolgreichem Login automatisch zurücklenkt.

Listing 2: EC2-Proxy in NodeJS

```
    // Load .env at startup
require('dotenv').config();

const express = require('express');
const morgan = require('morgan');
const { Pool } = require('pg');

// Sanitize and trim environment variables
const dbUser = process.env.DATABASE_USER?.trim();
const dbPass = process.env.DATABASE_PASSWORD?.trim();
const dbHost = process.env.DATABASE_HOST?.trim();
const dbPort = Number(process.env.DATABASE_PORT);
const dbName = process.env.DATABASE_NAME?.trim();

console.log('Starting DB proxy service');
console.log('CWD:', process.cwd());
console.log('Database config:', { host: dbHost, port: dbPort, user: dbUser, databa

// Set up Express
const app = express();

// 1. Morgan for HTTP logging
app.use(morgan(':method :url :status :res[content-length] - :response-time ms'));

// 2. JSON body parsing
app.use(express.json());

// 3. Sanity dump for POST bodies
app.use((req, res, next) => {
if (['POST','PUT','PATCH'].includes(req.method)) console.log('> BODY:', req.body);
next();
});

// 4. Postgres pool using sanitized env
const pool = new Pool({
host: dbHost,
port: dbPort,
user: dbUser,
password: dbPass,
database: dbName,
ssl: { rejectUnauthorized: false }
});

// Async wrapper helper
defaultWrapAsync = fn => (req, res, next) => fn(req, res, next).catch(next);

// 5. Health endpoint
defaultWrapAsync(async (req, res) => {
await pool.query('SELECT 1');
res.sendStatus(200);
});
app.get('/health', defaultWrapAsync(async (req, res) => res.sendStatus(200)));

// 6. Query endpoint
app.post('/query', defaultWrapAsync(async (req, res) => {
const { text, params } = req.body;
const { rows } = await pool.query(text, params);
res.json(rows);
}));

// 7. 404 catch-all
app.use((req, res) => res.sendStatus(404));

// 8. Error handler
```

Listing 3: Lambda-Funktion für die API

```
    import { Pool } from 'pg';

// Create a pool as a singleton so Lambda can reuse TCP connections
const pool = new Pool({
  host:     process.env.DB_HOST,
  database: process.env.DB_NAME,
  user:     process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  port:     process.env.DB_PORT,
  // If you need SSL, uncomment and adjust:
  // ssl: { rejectUnauthorized: false }
});

export const handler = async (event) => {
  const projectId = event.pathParameters?.projectId;
  if (!projectId) {
    return { statusCode: 400, body: 'Missing projectId path parameter' };
  }

  let client;
  try {
    client = await pool.connect();

    const result = await client.query(
      `SELECT "ProjectID", "AttributeID"
         FROM public."Newport_Project"
        WHERE "ProjectID" = $1`,
      [projectId]
    );

    if (result.rows.length === 0) {
      return { statusCode: 404, body: 'Project not found' };
    }

    return {
      statusCode: 200,
      headers: { 'Content-Type': 'application/json',
      'Access-Control-Allow-Origin': 'http://localhost:3000',
      'Access-Control-Allow-Credentials': true,
      // if you need cookies
      // add any other headers your client uses:
      'Access-Control-Allow-Headers': 'Authorization,Content-Type',},
      body: JSON.stringify(result.rows[0])
    };

  } catch (err) {
    console.error('DB error', err);
    return {
      statusCode: 500,
      body: 'Internal server error'
    };

  } finally {
    client?.release();
  }
};
```

7 Frontend-Implementierung

Das Frontend der Anwendung ist in drei logische Abschnitte eingeteilt. Zunächst übernimmt der Authentifizierungsbereich die Anmeldung und Speicherung benötiter Zugangsdaten. Daraus aufbauend erfolgt die Datenabfrage mithilfe der API, deren Ergebnisse dann im User Interface verarbeitet und dargestellt werden. Im Folgenden werden aufbau, Funktionsweise und Designentscheidungen der Anwendung näher beschrieben.

7.1 Architektur und Zuständigkeiten

Das Frontend der Anwendung wurde mit React und TypeScript umgesetzt und folgt einem komponentenbasierten Aufbau. Ziel der Architektur ist es, die Interaktion mit dynamisch generierten Attributen möglichst flexibel und intuitiv zu gestalten - unabhängig davon, wie viele oder welche Felder für ein bestimmtes Projektelement hinterlegt sind. Gleichzeitig soll die Schnittstelle gegenüber der REST-API möglichst schlank und leicht erweiterbar bleiben.

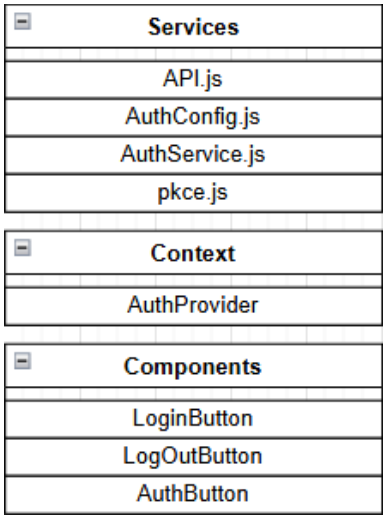
Funktional lässt sich die Anwendung in drei zentrale Teilbereiche aufteilen:

- **Authentifizierung:** Die Anmeldung erfolgt über einen OAuth2-Flow mit PKCE, der im Code über einen eigenen `AuthContext` abstrahiert ist. Dadurch stehen die Anmeldedaten allen Komponenten zentral zur Verfügung, ohne explizit durchgereicht werden zu müssen. Die Verbindung zur API wird über ein Auth-Token sichergestellt.
- **Dateninteraktion:** Operationen wie das Anlegen neuer Attribute, das Setzen von Werten oder deren Bearbeitung erfolgen über eine zentralisierte Service-Schicht, die alle API-Aufrufe kapselt. Fehlerbehandlung und Rückmeldung an das UI laufen ebenfalls darüber.
- **Dynamisches UI:** Die Oberfläche generiert auf Basis der zugrunde liegenden Attributdefinitionen automatisch passende Eingabefelder. Die Formulare reagieren dabei auf die Struktur und den Typ der zugewiesenen Attribute. Validierungen erfolgen sowohl im Frontend (Pflichtfelder, Typen) als auch über die API (Integritätsprüfungen, Typkonflikte).

Die Architektur wurde so aufgebaut, dass sie einerseits eine möglichst wartbare Codebasis ermöglicht und gleichzeitig flexibel genug bleibt, um auf unterschiedliche Nutzungsszenarien zu reagieren. Der Fokus liegt klar auf Funktionalität, nicht auf komplexem visuellem Design - gerade mit Blick auf interne Fachanwender:innen, die ohne technisches Vorwissen arbeiten.

7.2 Authentifizierung

Im Sinne der Erweiterbarkeit und Wartbarkeit wurde folgende Ordnerstruktur gewählt:



Der Login-Prozess wird in drei semantische Kategorien aufgeteilt. An erster Stelle stehen die Komponenten, die dem Nutzer die Interaktion mit der Authorisierungs-Logik ermöglichen.

7.2.1 Components

Obwohl theoretisch auch eine einzelne Komponente für den Login-Prozess ausreichen würde, wurde der Login/Logout-Button im Sinne der Übersichtlichkeit gedrittelt. Der Login- sowie Logout-Button sind sind in der Logik bis auf die ausgeführte Methode identisch: 5

Der Login-Button, welcher aus der Unternehmens-internen React-Bibliothek entnommen wird, ruft bei Interaktion die `login()`-Methode aus dem Authentifizierungs-Kontext auf, welcher im nächsten Abschnitt beschrieben wird.
Der gleiche Prozess findet beim Logout-Button statt, nur dass hier die `logout()`-Methode aufgerufen wird. 6

Diese beiden Komponenten werden dann im AuthButton kombiniert, wobei der Auth-Status entscheidet, welcher Button angezeigt wird: 7 Für die Entscheidung, welcher Button angezeigt wird, wird zuerst über die `useAuth()`-Hook[28] der Authentifizierungs-Kontext abgerufen. Der daraus resultierende Boolean-Wert `isAuthenticated` gibt an, ob der Nutzer eingeloggt ist oder nicht. Mit diesem Boolean-Wert wird dann entschieden, ob der Login- oder Logout-Button angezeigt wird. Das geschieht in diesem Fall mithilfe eines ternären Operators, der den Wert von `isAuthenticated` überprüft und abhängig vom Wert den entsprechenden Button zurückgibt.[29]

7.2.2 Context

Der Authentifizierungs-Kontext ist ein zentraler Bestandteil der Authorisierungs-Logik. Er stellt die Methoden `login()` und `logout()` zur Verfügung, ist aber zusätzlich auch für die Verarbeitung des Callbacks von Cognito zuständig. Zuerst werden für den Kontext die benötigten Hooks importiert und ein Context-Objekt erstellt: 8 9 Daraufhin werden für den Kontext relevante Variablen definiert, die den User, den Authentifizierungsstatus und die Tokens enthalten: Die `login()`-Methode generiert einen zufälligen State und einen PKCE-Verifier, speichert diese im Session Storage und leitet den Nutzer zur Authentifizierungs-URL weiter: 10 Die `logout()`-Methode löscht den Session Storage und leitet den Nutzer zur Logout-URL weiter: 11 Die `handleCallback()`-Methode wird aufgerufen, wenn der Nutzer nach der Authentifizierung zurück zur Anwendung geleitet wird. Das wird durch die `useEffect()`-Hook realisiert, die prüft, ob der Nutzer auf der Callback-Route ist und ob ein Code in der URL vorhanden ist. 12 Die `handleCallback()`-Methode extrahiert den Code und den State aus der URL, vergleicht den State mit dem im Session Storage gespeicherten Wert und tauscht den Code

[28] In JavaScript, speziell in React, ist eine Hook eine Funktion, mit der ein Zugriff auf React-Features wie State oder Lifecycle-Methoden in Funktionskomponenten möglich gemacht wird
[29] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator`

gegen Tokens aus. Falls der Austausch erfolgreich ist, werden die Tokens im Zustand gespeichert und der Nutzer wird zur Dashboard-Seite weitergeleitet: 13 Die `AuthProvider`-Komponente stellt den Authentifizierungs-Kontext für die gesamte Anwendung bereit: 14 Die `useAuth()`-Hook ermöglicht den Zugriff auf den Authentifizierungs-Kontext in anderen Komponenten der Anwendung.

7.2.3 Service

Der Service-Ordner enthält unterstützende Funktion für die Authorisierung, wie die Generierung des PKCE-Verifiers und -Challenges, den Austausch des Codes gegen Tokens und die Erstellung der Authentifizierungs-URL. Zuerst muss dafür die Konfiguration definiert werden, mit der gearbeitet wird.

7.2.4 AuthConfig

Die `authConfig.js`-Datei enthält die Konfiguration für die Authentifizierung, einschließlich der Endpunkte, Client-ID und Redirect-URI. Zusätzlich wird auch die Basis-URL der API definiert, um später API-Anfragen zu ermöglichen:15 Mit dieser Konfiguration wird nun gearbeitet, um alle weiteren benötigten Daten zu generieren.

7.2.5 AuthService

Der AuthService enthält drei Methoden, die für die Authorisierung benötigt werden:

- `buildAuthUrl()`: Diese Methode generiert die Authentifizierungs-URL, die den Nutzer zur Cognito-Anmeldeseite weiterleitet.
- `exchangeCodeForTokens()`: Diese Methode tauscht den erhaltenen Code gegen Tokens aus, die für die Authentifizierung und Autorisierung verwendet werden.
- `refreshTokens()`: Diese Methode aktualisiert die Tokens, wenn sie abgelaufen sind.

Die `buildAuthUrl()`-Methode erstellt die Authentifizierungs-URL, indem sie die Konfiguration und die PKCE-Challenge verwendet: 16 Die `exchangeCodeForTokens()`-Methode tauscht bei Rückruf von der Cognito-Authentifizierung den mitgegebenen Code für ein gültiges Auth-Token. Dieses kann dann verwendet werden, um API-Abfragen durchzuführen: 17 Dafür werden zuerst die nötigen Konfigurations-Daten geladen und die Parameter für die URL-Suche definiert. Mit diesen Daten wird dann eine Anfrage an den TokenEndpoint gesendet und die Antwort als das Token zurückgegeben, sollten keine Fehler auftreten.

7.2.6 PKCE

PKCE(Proof Key for Code Exchange) ist eine Erweiterung des OAuth 2.0-Protokolls, welches zusätzliche Sicherheit für Clients verspricht, die nicht in der Lage sind, ihre Client-Geheimnisse sicher zu speichern. Innerhalb dieser Anwendung wird PKCE verwendet, um den Authentifizierungsprozess sicherer zu gestalten. Dies geschieht über zwei Methoden, `generateVerifier()` und `generateChallenge()`, die jeweils den Verifier und die Challenge generieren.

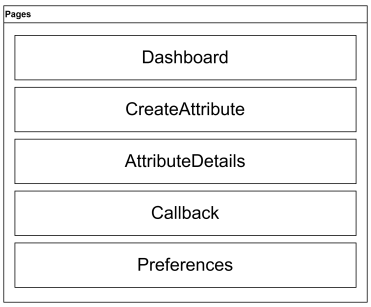
Die `generateVerifier()`-Methode generiert einen zufälligen Verifier, der als Basis für die Challenge dient. Das geschieht, indem ein Array von 32 zufälligen Bytes generiert wird, welches dann in einen hexadezimalen String umgewandelt wird: 18 Die `generateChallenge()`-Methode nimmt den Verifier als Eingabe und generiert die Challenge, indem der Verifier in einen SHA-256-Hash umgewandelt wird. Das Ergebnis wird dann in einen Base64-URL-kodierten String umgewandelt: 19

7.2.7 API

Der API-Service ist die Hook `useApi()`, welche für die Kommunikation mit der Backend-API zuständig ist. Sie Kapselt die Methode `callAPI()` ab, welche den Zugriff auf die Konfigurierte API ermöglicht: 20 Die Daten, die aus der `callApi()`-Methode zurückgegeben werden, können dann in den Komponenten verwendet werden, um die Daten anzuzeigen oder zu verarbeiten.

7.3 Dynamische Formularlogik und User Interface

Das User Interface der Anwendung ist in mehrere Komponenten unterteilt, die jeweils für verschiedene Teile der Anwendung zuständig sind:



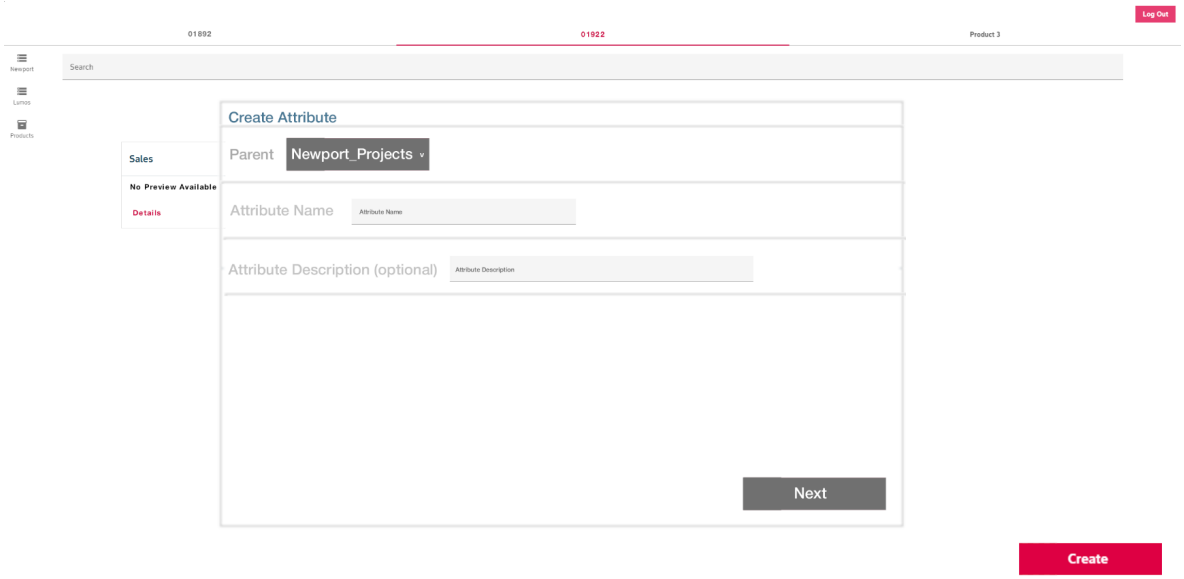
Den Kern der Anwendung bildet die `Dashboard`-Komponente, welche die Hauptansicht der Anwendung beinhaltet. Von hier aus werden die verscheidenen Unterseiten aufgerufen (mit Ausnahme der Callback-Seite, die nur für die Authorisierung zuständig ist).

7.4 Anwendungsbeispiel

Im Folgenden soll anhand eines konkreten Anwendungsbeispiels die Funktionsweise der Anwendung verdeutlicht werden. Dabei soll gezeigt werden, wie ein Attribut erstellt, zugeordnet, bearbeitet und gelöscht werden kann. Dadurch soll die Funktionsweise des Systems aus Sicht eines Endnutzers erklärt werden.

7.4.1 Anlegen eines neuen Attributs

Um ein neues Attribut anzulegen, navigiert der Nutzer zur `Attributserstellung`, indem er den entsprechenden Button im Dashboard klickt. Daraufhin wird der Nutzer aufgefordert, die erforderlichen Informationen für das Attribut einzugeben. Dabei wird im Hintergrund durch die Anwendung die Konformität der Eingaben überprüft, und gegebenenfalls eine Fehlermeldung angezeigt. Durch klicken des `Next`-Buttons kann der Nutzer dann die Eingaben bestätigen und die zweite Seite der Attributserstellung aufrufen.



Die zweite Seite der Attributserstellung ermöglicht es dem Nutzer, dem Attribut sofort Werte hinzuzufügen. Dafür werden zuerst Informationen über die Art der Werte abgefragt, woraufhin der Nutzer die Werte in einer Tabelle eingeben kann. Durch klicken des `Create`-Buttons kann der Nutzer dann seine Eingaben bestätigen, und das Attribut wird erstellt. Im Hintergrund wird dafür die API aufgerufen, die das Attribut in der Datenbank anlegt.

7.4.2 Bearbeiten eines Werts

Um einen Wert zu bearbeiten, navigiert der Nutzer zuerst auf die Detailansicht des Attributs, in welchem der Wert enthalten ist. Dort findet er eine Tabelle vor, die alle Werte aufführt, und ihm ermöglicht, die Werte zu bearbeiten. Durch klicken des `Save`-Buttons kann der Nutzer dann seine Änderungen speichern. Im Hintergrund wird dafür die API aufgerufen, die den Eintrag in der Datenbank aktualisiert.

7.4.3 Löschen eines Attributs

Um ein Attribut zu löschen, navigiert der Nutzer auf die Detailansicht des Attributs, welches er löschen möchte. Dort findet er einen `Delete`-Button, der ihn auffordert, das Löschen zu bestätigen. Nach der Bestätigung wird das Attribut gelöscht, und der Nutzer wird auf die Startseite zurückgeleitet. Im Hintergrund wird dafür die

API aufgerufen, die den Eintrag und alle zugehörigen Referenzen in der Datenbank löscht.

7.4.4 Zusammenfassung der Attributverwaltung

Das User Interface ist darauf ausgelegt, dem Nutzer eine einfache und intuitive Möglichkeit zu bieten, Attribute zu erstellen, zu bearbeiten und zu löschen. Dafür soll der Nutzer in möglichst wenigen Schritten zu seinem Ziel gelangen, und die Anwendung soll ihn aktiv unterstützen, indem sie im Hintergrund möglichst viele Aufgaben übernimmt. Beispielsweise wird die Zugehörigkeit zu einem semantischen Projekt automatisch erkannt, und der Nutzer muss dieses nicht manuell angeben.

7.5 Herausforderungen und Entscheidungen

Während der Umsetzung des Frontends sind verschiedene Herausforderungen aufgetreten, die sich nicht nur auf den Code selbst, sondern auch auf Nutzerführung, Fehlerverhalten und Wartbarkeit ausgewirkt haben. Im Folgenden sollen die wichtigsten dieser Punkte - und die Entscheidungen, die daraus resultierten - kurz beschrieben werden.

Dynamisches UI ohne Sonderfälle

Eine der zentralen Fragen war: Wie lässt sich ein Interface bauen, das automatisch auf neue Attribute reagiert, ohne dass man bei jeder kleinen Änderung im Code nachbessern muss? Die Lösung bestand darin, Eingabeformulare komplett dynamisch zu generieren - also zur Laufzeit auf Basis der hinterlegten Attribut-Definitionen. Das macht den Code insgesamt generischer, aber auch schwerer lesbar. Trotzdem war es die bessere Entscheidung, da so alle Objekte (Formulierungen, Segmente etc.) einheitlich behandelt werden können. Anpassungen am Layout einzelner Felder wurden bewusst vermieden, um die Wartbarkeit nicht zu gefährden.

Validierung: sofort oder später?

Ursprünglich war angedacht, die Validierung ausschließlich über das Backend laufen zu lassen - also Fehler erst nach einem Request zurückzumelden. Das führte aber zu Frust, weil Nutzer:innen gar nicht wussten, ob ihre Eingaben korrekt sind. Deshalb wurde zusätzlich eine einfache Validierung im Frontend umgesetzt: Pflichtfelder, Datentypen und leere Eingaben werden direkt geprüft, bevor überhaupt ein Request abgeschickt wird. Dadurch werden Fehler früher abgefangen, ohne dass die API komplett entlastet werden muss.

Generik trifft auf Lesbarkeit

Weil viele Teile des Frontends dynamisch funktionieren müssen, ist der Code an manchen Stellen weniger intuitiv. Zum Beispiel müssen nicht nur Felder, sondern auch zugehörige State-Objekte, Event-Handler und API-Requests zur Laufzeit angepasst werden. Um das im Griff zu behalten, wurden klare Zuordnungen zwischen Attributtyp und UI-Komponente eingeführt - z. B. dass „Zahl“ immer ein bestimmtes Input-Feld erzeugt. Solche Regeln wurden in Hilfsfunktionen ausgelagert, damit die Hauptkomponenten übersichtlich bleiben.

Fehlertoleranz bei API-Problemen

Da das Frontend stark auf die API angewiesen ist, war schnell klar: Wenn eine Anfrage scheitert oder langsam ist, muss das UI trotzdem stabil bleiben. Deshalb wurden für alle Requests Lade- und Fehlerzustände eingebaut, die direkt im UI angezeigt werden. Das heißt: Buttons deaktivieren sich automatisch bei laufender Anfrage, Fehler werden in Klartext angezeigt - und das System bleibt insgesamt bedienbar, auch wenn das Backend mal nicht reagiert.

7.6 Fazit zur Frontend-Implementierung

Im Rahmen des Frontends wurde eine benutzerfreundliche und sichere Anwendung entwickelt, die es Nutzern ermöglicht, Attribute zu erstellen, zu bearbeiten und zu löschen. Um die Erweiterbarkeit und Wartbarkeit der Anwendung zu gewährleisten,

wurde eine modulare Struktur gewählt, durch welche neue Funktionen und Komponenten einfach hinzugefügt werden können. Die Anwendung ist so konzipiert, dass Nutzer einfach und intuitiv mit ihr interagieren können, während im Hintergrund komplexe Logik und Sicherheitsmaßnahmen abgearbeitet werden. Dafür wurden folgende Features implementiert:

- Dynamische Attributerstellung
- Zuweisung von Attributen zu Datenobjekten
- Werteingabe und Validierung
- Bearbeiten und Löschen von Werten
- Dynamisches Rendering von Formularen
- API-Kommunikation mit Lade- und Fehlerstatus
- Zustandsverwaltung mit React Hooks
- Grundlegende Benutzerführung
- Selbsterklärende Struktur und Modularität

8 Evaluation

Im vorherigen Kapitel wurde die Umsetzung des Frontends der Anwendung beschrieben. Dabei wurde der Fokus auf die Möglichkeit gelegt, dynamische Attribute zu erstellen, zuzuordnen und zu verwalten, ohne dabei die feste Datenbankstruktur der ursprünglichen Tabelle verändern zu müssen.

Im Folgenden soll geprüft werden, ob die zuvor definierten funktionalen Anforderungen (**??**) vollständig umgesetzt werden konnten, und ob das System letztendlich den praktischen Nutzen bringt, der von den Auftraggebenden erwartet wird.

Im Fokus steht dabei nicht die Auswertung technischer Metriken, sondern die Frage, die für die Nutzer am relevantesten ist: Funktioniert die Anwendung in der alltäglichen Nutzung so, wie es gedacht ist?

Dafür wird zuerst ein Abgleich mit den Anforderungen getätigt, wonach bewertet wird, wie sich die Anwendung in typischen Szenarien verhält. Abschließend folgt eine Einschätzung, inwieweit die Anwendung wartbar und erweiterbar ist.

8.1 Funktionale Zielerreichung

Die Planung und Umsetzung der Anwendung basiert auf den in Kapitel 3.2 definierten „Must-Have“-Anforderungen. Alle für die Anwendung zentralen Funktionen wurden implementiert und erfolgreich getestet: Zur Bewertung der funktionalen Zielerreichung werden drei KPIs verwendet, durch welche zentrale Interaktionen abgebildet werden:

- **Attributerstellung:** Ein Nutzer ist in der Lage, Attribute mit Titel, Beschreibung und Typ zu erstellen. Die dafür bereitgestellten Eingabefelder werden dynamisch validiert, sodass nur gültige Eingaben möglich sind.
- **Zuweisung und Eingabe:** Attribute sind spezifischen Tabellen zuweisbar, und Werte eines Attributs können unter Berücksichtigung des Kontexts erfasst werden.
- **Löschung:** Einzelne Werte (solange noch weitere vorhanden sind) können gelöscht werden, ohne das Attribut selbst entfernen zu müssen. Attribute können

vollständig gelöscht werden, ohne die referentielle Integrität der Datenbank zu gefährden.

- **Benutzerführung:** Das User Interface bietet im Hintergrund Validierung, bei Bedarf Fehleranzeigen und Ladezustände für alle Interaktionen, um eine intuitive Nutzererfahrung zu gewährleisten.

Die Anwendung erfüllt den funktionalen Zweck, für den sie konzipiert wurde, vollumfänglich. Eine Bedienung der Anwendung ist auch ohne technische Vorkenntnisse über dynamische Attribute möglich.

8.2 Reaktion auf typische Nutzungsszenarien

Im Rahmen des Testbetriebs wurde das System mehreren Standard-Szenarien unterzogen:

- Erstellung eines neuen Attributs mit anschließendem Wertimport
- Bearbeiten eines bestehenden Werts mit falschem Format
- Zeitgleiches Editieren mehrerer Einträge
- Netzwerkunterbrechung während einer Operation

Alle Szenarien wurden durch die Anwendung korrekt abgefangen: Fehler wurden, wenn gegeben, angezeigt, Nutzerinteraktionen wurden sauber zurückgerollt oder blockiert, und Ladezustände wurden angezeigt.

Die Anwendung verhält sich damit auch unter erschwerten Bedingungen robust und transparent.

8.3 Leistungsfähigkeit der API

Alle zentralen Funktionen der Anwendung basieren auf API-Anfragen, weshalb die Leistungsfähigkeit der API elementar für die Leistungsfähigkeit der Anwendung ist. Dies steht insbesondere im Fokus, da die Leistungsfähigkeit einiger Endgeräte der nutzenden Personen stark eingeschränkt ist, wodurch die Antwortzeiten der API-Anfragen direkt

die Nutzererfahrung beeinflussen. Die Leistungsfähigkeit soll anhand von vier KPIs bewertet werden:

- **Median-Response-Time:** Die Zeit in Millisekunden, die vom API-Aufruf bis zur Antwort verstreicht, soll unter 500ms liegen.
- **Durchsatzrate:** Die Anzahl API-Requests, die stabil verarbeitet werden können, soll nach Möglichkeit doppelt so hoch sein wie die erwartete maximale Last. Diese umfasst in der momentanen Planung 20 Anfragen pro Sekunde [30].
- **Fehlerquote unter Last:** Die Fehlerquote der API bei 100 Requests soll unter 5% liegen.
- **Datenbank-Roundtrips pro Anfrage:** Eine API-Anfrage soll nicht mehr als 2 Queries an die Datenbank stellen.

Im Folgenden wird die Erfüllung dieser KPIs untersucht, beginnend mit der Median-Response-Time.

8.3.1 Median-Response-Time

Um die Median-Response-Time zu messen, wurde eines der API-Requests, welches regelmäßig aufgerufen wird, mit einer Zeitmessung ausgestattet. Dafür wurde vor und nach dem Request die aktuelle Zeit abgerufen, und die Differenz als Response-Time ausgegeben.

Listing 4: Median Response Time Testing

```
const start = performance.now();
try{
    const result = await callApi('/projects/fetchProject/{projectId
        }');
    const end = performance.now();

    console.log(' Response Time: ${Math.round(end - start)} ms');
    .
    .
    .
}
```

[30] 2 Requests pro Sekunde pro Nutzer, 10 Key User sind eingeplant

Das Ergebnis der Messung ist in Abbildung **??** zu sehen. Die Messung wurde über 1000 Anfragen durchgeführt, um repräsentative Werte garantieren zu können. Im Schnitt liegt dabei die Response-Time bei 122ms, was deutlich unter der angestrebten Grenze von 500ms liegt. Da die Datenbank noch vergleichsweise klein ist, ist eine Abweichung nach oben in Zukunft zu erwarten, jedoch sollte die Grenze von 500ms auch bei einer größeren Datenbank nicht überschritten werden.

8.3.2 Durchsatzrate

Die Durchsatzrate wurde getestet, indem manuell 1000 Anfragen an die API in schneller Sukzession gesendet wurden. Dabei wurde die Fehlerrate, Gesamtzeit und die Anzahl der Anfragen, die pro Sekunde verarbeitet wurden, gemessen. Die Ergebnisse sind in Abbildung **??** zu sehen. Die Durchsatzrate liegt bei 7.29 Requests pro Sekunde, und liegt somit deutlich unter dem Zielwert von 20 Requests pro Sekunde. In Zukunft muss also die Durchsatzrate der API erhöht werden, um die prognostizierte Last effizient zu bewältigen.

8.3.3 Fehlerquote unter Last

Die Fehlerquote unter Last wurde ebenfalls im Rahmen des Tests der Durchsatzrate gemessen. Dabei wurde die Anzahl der fehlerhaft beantworteten Anfragen gezählt, woraus eine Fehlerquote berechnet wurde. Die Ergebnisse sind in Abbildung **??** zu sehen. Die Fehlerquote liegt mit 0% deutlich unter der angestrebten Grenze von 5%.

8.3.4 Fazit zur Leistungsfähigkeit der API

Die Leistungsfähigkeit der API ist insgesamt als ausreichend zu bewerten. Zwar liegt die Durchsatzrate unter dem angestrebten Zielwert, da dieser jedoch stark über der maximal erwartbaren Last liegt, ist die API dennoch in der Lage, die erwartbare Last zu bewältigen. Alle weiteren Metriken liegen deutlich im Rahmen der angestrebten Werte. Um die API vollständig zukunftssicher zu gestalten, muss die Durchsatzrate jedoch erhöht werden. Dies kann hauptsächlich durch Caching von häufig angefragten Daten erreicht werden, um die Anzahl der Datenbankabfragen zu reduzieren.

8.4 Wartbarkeit und Erweiterbarkeit der Anwendung

Die Frontend-Architektur ist modular aufgebaut, wodurch sauber zwischen Logik, Darstellung und API-Kommunikation getrennt werden kann. Neue Komponenten lassen sich mit vertretbarem Aufwand ergänzen, ohne neue Fehler in bestehenden Komponenten zu verursachen.

Die Kommunikation im Backend erfolgt über eine gekapselte Service-Schicht, die leicht angepasst oder erweitert werden kann, sollte sich die Struktur der API verändern. Grundlegende Strukturen müssen demnach nicht angepasst werden, um das System zu erweitern.

Auch die Datenbankstruktur ermöglicht eine schrittweise Erweiterung, unter anderem über die Spezifikations-Tabelle, die eine hohe Flexibilität für die Informationen , die über Attribute gespeichert werden können, bietet.

8.5 Zusammenfassung der Evaluation

Die Anwendung erfüllt sowohl die funktionalen als auch technischen Anforderungen vollumfänglich. Sie ist stabil im Betrieb, auch für nicht-technische Nutzer nachvollziehbar und lässt sich technisch gut warten und erweitern. Die Zielsetzung der Arbeit - ein erweiterbares Framework zur dynaamischen Attributverwaltung - wurde somit erreicht.

9 Fazit

Im Rahmen dieser Bachelor-Arbeit sollte ein technisches Framework entwickelt werden, mit welchem es möglich ist, benutzerspezifische Informationen dynamisch zu bestehenden Datensätzen zu speichern, ohne die zugrunde liegende Datenstruktur anpassen zu müssen.

Im Zentrum stand dabei die Frage, inwieweit diese Flexibilität einen Einschnitt in die Integrität der Daten, die Benutzerfreundlichkeit und Wartbarkeit des Systems darstellt. Die Ergebnisse der Arbeit zeigen, dass eine solche Lösung technisch realisierbar und im praktischen Betrieb stabil einsetzbar ist.

Sowohl das Datenmodell als auch das Frontend wurden so konzipiert, dass Erweiterungen möglich sind, ohne dabei bestehende Strukturen verändern zu müssen. Der modulare Aufbau der Anwendung ermöglicht es, neue Komponenten mit überschaubarem Aufwand zu integrieren. Besonderer Fokus lag auf einer klaren Trennung zwischen API-Kommunikation, Benutzeroberfläche und der Geschäftslogik - wodurch die Wartbarkeit langfristig gesichert wird.

In der Evaluation konnte gezeigt werden, dass alle funktionalen Kernanforderungen erfüllt wurden. Auch unter Last und nicht-idealen Bedingungen (z. B. langsamer Verbindung oder inkorrekter Nutzereingaben) zeigt sich die Anwendung robus und nutzbar.

Trotzdem bleiben weiterhin offene Punkte: Die Durchsatzrate der API ist noch nicht ausreichend, um die prognostizierte maximale Last vollständig zu bewältigen. Darüber hiknaus ist langfristig ein Rollen- und Rechtesystem erforderlich, um den Schutz sensibler Daten zu gewährleisten.

Insgesamt zeigt diese Arbeit, dass dynamische Erweiterbarkeit und technische Robustheit kein WIderspruch sein müssen, solange Struktur, Validierung und Nutzerführung in der Architektur von Beginn an berücksichtigt werden.

Anhang

Anhangsverzeichnis

Anhang 1 Gesprächsnotizen

Anhang 1.1 Gespräch mit Werner Müller

Gespräch mit Werner Müller am 01.01.2013 zum Thema XXX:

- Über das gute Wetter gesprochen
- Die Regenwahrscheinlichkeit liegt immer bei ca. 3%
- Das Unternehmen ist total super
- Hier könnte eine wichtige Gesprächsnotiz stehen

Anhang 2 Abbildungen

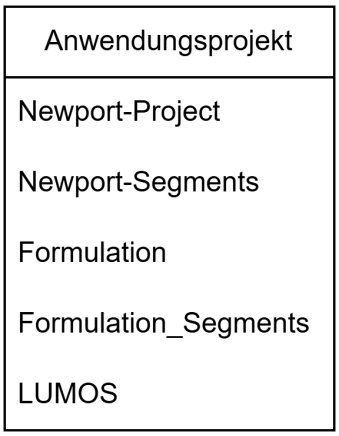


Abbildung 2: projektGrafik

Anhang 3 Listings

Listing 5: JS-Code for the LogIn-Button

```
// components/LoginButton.js
import React from 'react';
import { useAuth } from '../context/AuthContext';
import { Button } from '@element/react-components';

export default function LoginButton() {
    const { login } = useAuth();

    return <Button onClick={login} label="Log In"/>
}
```

Listing 6: JS-Code for the LogOut-Button

```
// components/LogoutButton.js
import React from 'react';
import { useAuth } from '../context/AuthContext';
import { Button } from '@element/react-components';

export default function LogoutButton() {
    const { logout } = useAuth();

    return <Button onClick={logout} label="Log Out"/>
}
```

Listing 7: JS-Code for the AuthButton

```
// components/AuthButton.js
import React from 'react';
import { useAuth } from '../context/AuthContext';
import LoginButton from './LoginButton';
import LogoutButton from './LogoutButton';

export default function AuthButton() {
    const { isAuthenticated } = useAuth();

    return isAuthenticated ? <LogoutButton /> : <LoginButton />;
}
```

Listing 8: Import und Kontext

```
import React, { createContext, useState,
useEffect, useContext } from 'react';
import { useNavigate, useLocation } from 'react-router-dom';
import {buildAuthUrl, generateChallenge, generateVerifier,
exchangeCodeForTokens} from '../service/authService';
import authConfig from '../services/authConfig';
import {generateVerifier, generateChallenge} from '../service/pkce'
    ;

const AuthContext = createContext();
```

Listing 9: Variablen für den Authentifizierungs-Kontext

```
const [user, setUser] = useState(null);
const [isAuthenticated, setIsAuthenticated] = useState(false);
const [tokens, setTokens] = useState(null);
const navigate = useNavigate();
const location = useLocation();
```

Listing 10: Login-Methode

```
async function login() {
    const state = crypto.randomUUID();
    const verifier = generateVerifier();
    const challenge = await generateChallenge(verifier);

    sessionStorage.setItem('oauth_state', state);
    sessionStorage.setItem('pkce_verifier', verifier);

    const url = buildAuthUrl({state, code_challenge: challenge});

    window.location.href = url;
}
```

Listing 11: Logout-Methode

```
function logout() {
    sessionStorage.clear();
    const {logoutEndpoint, clientId, logoutUri} =authConfig;
    window.location.href = `${logoutEndpoint}?client_id=${clientId
        }&logout_uri=${encodeURIComponent(logoutUri)}`;
}
```

Listing 12: Callback-Handling

```
useEffect(() => {
    if (location.pathname === '/callback' && location.search.
        includes('code=')) {
        handleCallback();
    }
}, [location, handleCallback]);
```

Listing 13: Callback-Handling

```
async function handleCallback() {
    const params = new URLSearchParams(location.search);
    const code = params.get('code');
    const state = params.get('state');
    const saved = sessionStorage.getItem('oauth_state');

    if(!code || !state || state !== saved) {
        return navigate('/', {replace: true});
    }

    try{
        const verifier = sessionStorage.getItem('pkce_verifier');
        const tokenSet = await exchangeCodeForTokens(code, verifier
            );
        setTokens(tokenSet);

        const [, payload] = tokenSet.id_token.split('.');
        const userInfo = JSON.parse(atob(payload));
        setIsAuthenticated(true);

        sessionStorage.removeItem('pkce_verifier');
        sessionStorage.removeItem('oauth_state');

        navigate('/dashboard', {replace: true});
    } catch (err) {
        console.error('Error during authentication:', err);
        navigate('/', {replace: true});
    }
}
```

Listing 14: AuthProvider-Komponente

```
        return (
            <AuthContext.Provider
            value={{ user, isAuthenticated, tokens, login, logout }}>
                {children}
            </AuthContext.Provider>
        );
    }
    export function useAuth() {
        return useContext(AuthContext);
    }
```

Listing 15: AuthConfig

```
        const domain = process.env.REACT_APP_COGNITO_DOMAIN;
        const clientId = process.env.REACT_APP_COGNITO_CLIENT_ID;
        const redirectUri = process.env.REACT_APP_COGNITO_REDIRECT_URI;
        const logoutUri = process.env.REACT_APP_COGNITO_LOGOUT_URI;
        const apiBaseUrl = process.env.REACT_APP_API_BASE_URL;
        export const authConfig = {
            domain,
            clientId,
            redirectUri,
            logoutUri,
            apiBaseUrl: apiBaseUrl,
            authEndpoint: `https://${domain}/oauth2/authorize`,
            tokenEndpoint: `https://${domain}/oauth2/token`,
            logoutEndpoint: `https://${domain}/logout`,
            responseType: 'code',
            scope: 'openid profile email',
        };
```

Listing 16: Build-Auth Methode

```
export function buildAuthUrl({state, code_challenge}) {
    const {
        authorizeEndpoint,
        clientId,
        redirectUri,
        responseType,
        scope
    } = authConfig;
    const params = new URLSearchParams({
        client_id: clientId,
        redirect_uri: redirectUri,
        response_type: responseType,
        scope,
        state,
        code_challenge_method: 'S256',
        code_challenge
    });
    return '${authorizeEndpoint}?{params}';
}
```

Listing 17: ExchangeCodeForToken Methode

```
export async function exchangeCodeForToken(code, code_verifier){
    const {tokenEndpoint, clientId, redirectUri} = authConfig;

    const params = new URLSearchParams({
        grant_type: 'authorization_code',
        client_id: clientId,
        code,
        redirect_uri: redirectUri,
        code_verifier
    });

    const resp = await fetch(tokenEndpoint, {
        method: 'POST',
        headers: {'Content-Type': 'application/x-www-form-
            urlencoded' },
        body: params.toString()
    });

    const text = await resp.text();

    if(!resp.ok) throw new Error('Token Exchange failed: $(text)')
    return JSON.parse(text);
}
```

Listing 18: PKCE-Verifier-Generierung

```
export function generateVerifier() {
    const array = new Uint8Array(32);
    crypto.getRandomValues(array);
    return Array.from(array, b => ('0'+ b.toString(16)).slice(-2)).
        join('');
}
```

Listing 19: PKCE-Challenge-Generierung

```
export async function generateChallenge(verifier) {
    const encoder = new TextEncoder();
    const data = encoder.encode(verifier);
    const hash = await crypto.subtle.digest('SHA-256', data);
    return btoa(String.fromCharCode(...new Uint8Array(hash)))
        .replace(/\+/g, '-').replace(/\//g, '_').replace(/=+$/, '')
            ;
}
```

Listing 20: PKCE-Challenge-Generierung

```
async function callApi(path, options = {}){
    const url = '${authConfig.apiBaseUrl}${path}';
    const resp = await fetch(url, {
        ...options,
        headers: {
            'Content-Type': 'application/json',
            Authorization: 'Bearer ${tokens.access_token}',
            ...options, headers
        }
    });
    if (!resp.ok) throw
    new Error('API error ${resp.status}: ${await resp.text()}');
    return resp.json();
}
```

Quellenverzeichnis

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorthesis selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Langenfeld, 10.06.2025

Thomas Benjamin Hopf

Literatur

Codd, E. F. (1970). „A Relational Model of Data for Large Shared Data Banks“. In: *Communications of the ACM* 13.6, S. 377–387. DOI: 10.1145/362384.362685. URL: https://rebelsky.cs.grinnell.edu/Courses/CS302/2007S/Readings/codd-1970.pdf.

Date, C. J. (2003). *An Introduction to Database Systems*. 8. Aufl. Addison-Wesley.

Erdem, H. und Finin, T. (2009). „Supporting dynamic attributes: An EAV variant for metadata-driven databases“. In: *Proceedings of the 2nd International Conference on Advances in Databases, Knowledge, and Data Applications*. Springer, S. 45–56. DOI: 10.1007/978-3-540-25949-2_5.

Galletta, D. F., Henry, R. M., McCoy, S. und Polak, P. (2006). „When the wait isn't so bad: The interacting effects of website delay, familiarity, and breadth“. In: *Journal of the Association for Information Systems* 7.1, S. 3–28. DOI: 10.17705/1jais.00107.

Leavitt, Nathan (2010). „Platform as a service“. In: *Computer* 43.2, S. 4–6. DOI: 10.1109/MC.2010.58.

Li, C. und Box, D. (2005). „A metadata-driven EAV database design to support dynamic attributes in web applications“. In: *Journal of Database Management* 16.4, S. 1–23. DOI: 10.4018/jdm.2005100101. URL: https://www.sciencedirect.com/science/article/pii/S1532046405000502.

Nadkarni, P. M., Marenco, L., Chen, R., Skoufos, E. und Shepherd, G. M. (2001). „Organization of heterogeneous scientific data using the EAV/CR representation“. In: *Journal of the American Medical Informatics Association* 8.1, S. 217–236. DOI: 10.1136/jamia.2001.0080217. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC61391/.

Nielsen, Jakob (2014). *Placeholders in form fields are harmful*. URL: https://www.nngroup.com/articles/form-design-placeholders/.

Nielsen, Jakob (2016). *Error message guidelines*. URL: https://www.nngroup.com/articles/error-message-guidelines/.

OWASP Foundation (2017). *OWASP Top Ten 2017: The Ten Most Critical Web Application Security Risks*. URL: https://owasp.org/www-project-top-ten/2017/.

Parnas, David L. (1972). „On the criteria to be used in decomposing systems into modules“. In: *Communications of the ACM* 15.12, S. 1053–1058. DOI: 10.1145/361598.361623.

Prasad, U. (März 2025). „SQL and the relational model: Enduring standards in the age of AI“. In: *Dataversity*. URL: https://www.dataversity.net/sql-and-the-relational-model-enduring-standards-in-the-age-of-ai/.