Erweiterung der Datentypen in der Cloudanwendung QDAcity

MASTER THESIS

Lara Sulzbach

Eingereicht am 30. März 2026

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open Source Software

Betreuer:
Andreas Kaufmann PhD
Prof. Dr. Dirk Riehle, M.B.A.

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 30. März 2026

Lizenz

Erlangen, 30. März 2026

Abstract

Data triangulation is a well-established method in qualitative research projects fostering the synthesis of theories by comparing different perspectives from a variety of data sources. This widely used approach poses a challenge for Qualitative Data Analysis tools like QDAcity because different data types need to be supported. Usually, each of these data types requires its own UI and data model while access to the same code book and collaboration within the project team must be sustained. This master thesis addresses the challenge by integrating spreadsheet support while maintaining all collaborative workflows using Conflict-free Replicated Data Types (CRDTs). The new XLSX- and DOCX-file import enables a seamless transition from office tools to QDAcity generating a user-friendly experience. With these adjustments, QDAcity is becoming better aligned with the standards users are familiar with from popular text editing and spreadsheet software tools and the support of projects with a diverse set of data types is extended.

Zusammenfassung

Datentriangulation ist eine in qualitativen Forschungsprojekten weit verbreitete Methode zur Ableitung einer Theorie aus den verschiedenen Perspektiven unterschiedlicher Quellen. Dieser Ansatz stellt für Software zur Qualitativen Datenanalyse wie QDAcity eine Herausforderung dar, weil eine Vielzahl von Datentypen unterstützt werden muss. Üblicherweise erfordert jeder dieser Datentypen aufgrund seiner jeweiligen Eigenarten eine spezifische Benutzeroberfläche und ein eigenes Datenmodell. Gleichzeitig ist aber für alle Datentypen der Zugriff auf ein übergreifendes Codebuch notwendig und die Möglichkeit zur kollaborativen Zusammenarbeit innerhalb eines Projektteams muss sichergestellt sein. Diese Arbeit widmet sich dieser Herausforderung, indem sie einerseits die Unterstützung von Tabellenkalkulationsdokumenten integriert und dabei ein kollaboratives Arbeiten mittels Conflict-free Replicated Data Types (CRDTs) gewährleistet. Anderseits wird durch die neue Import-Funktion für XLSX- und DOCX-Dateien einen reibungslosen Übergang von populären Office-Tools zu QDAcity ermöglicht, was die Benutzerfreundlichkeit signifikant erhöht. Durch diese Anpassungen nähert sich QDAcity an die vom Nutzer gewohnten Standards üblicher Textverarbeitungs- und Tabellenkalkulationssoftware an und erweitert die Unterstützung für Projekte mit unterschiedlichen Datentypen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Akronyme

GCP Google Cloud Plattform

GCS Google Cloud Storage

CES Collaborative-Editing-Service

XSS Cross-Site-Scripting

WCAG Web Content Accessiblity Guidelines

CRDTs Conflict-free Replicated Data Types

1 Einleitung

In der Wissenschaft bedeutet Triangulation, vereinfacht ausgedrückt, dass ein Forschungsgegenstand aus mehreren Blickwinkeln betrachtet wird (Flick, 2008). Die Datentriangulation als spezielle Form der Triangulation zeichnet sich laut Kuckartz (2014) durch das Einbeziehen unterschiedlicher Datenquellen aus und ist eine in der qualitativen Forschung beliebte Methodik, um ein Phänomen mit Hilfe verschiedenartiger Datensätze aus mehreren Perspektiven zu betrachten. Damit eine Software für qualitative Datenanalyse diese Methodik unterstützen kann, muss sie eine breite Palette von Quellen verarbeiten können. Die Herausforderung dieser Anforderung ist, dass die unterschiedlichen Datentypen zwar aufgrund ihrer Eigenschaften sowohl eigene Benutzeroberflächen zum Anzeigen und Bearbeiten des Dokuments als auch eigene Datenstrukturen zum Persistieren benötigen, gleichzeitig aber übergreifende, einheitliche Strukturen für die qualitative Datenanalyse brauchen. Idealerweise sollen die Dokumente unabhängig von ihrem Datentyp innerhalb eines Projekts auch kollaborativ bearbeitet werden können. Ziel dieser Masterarbeit ist die Unterstützung von Projekten mit Datentriangulation in der Cloudanwendung QDAcity[1] durch das Einbinden neuer Datentypen zu verbessern. Die Webanwendung QDAcity, die aus einem Forschungsprojekt der Friedrich-Alexander-Universität Erlangen-Nürnberg am Lehrstuhl für Open-Source-Software entstand, ist eine Software zur qualitativen Datenanalyse. Für die Analyse kann der Nutzer eigene Dateien hochladen, um diese zu codieren, indem theoretische Konstrukte in den Daten identifiziert und mit Codes markiert werden (Kaufmann et al., 2023). Das Hochladen von Dateien ist dabei bisher auf die Datentypen RTF, TXT, PDF und Audioformate beschränkt. Mit den Datenformaten DOCX und XLSX von Microsoft fehlen zwei der am weitesten verbreiteten Dokumentformate in den Upload-Optionen. Für eine qualitative Datenanalyse in einer DOCX-Datei musste ein Nutzer diese bisher erst in eine RTF-Datei umwandeln, um diese dann hochladen zu können, wobei nur die Verarbeitung von sehr einfachen Textdokumenten ohne eingebettete Bilder, Tabellen und Links möglich war. Für XLSX-Dateien blieb nur die Möglichkeit, diese als PDF hochzuladen, weil QDAcity bisher keine Tabellenkalkulationsdokumente unterstützte. Die Umwandlung in das PDF-Format verhindert aber eine weitere

[1] https://qdacity.com/de/

Bearbeitung des Dokuments. Diese, im Hinblick auf Datentriangulation unzufriedenstellende Situation, wurde im Rahmen der Masterarbeit verbessert, wobei sich die Arbeit auf die Datentypen DOCX, RTF und XLSX konzentrierte. Für Textdokumente wurde eine Unterstützung von DOCX-Dateien implementiert und der vorhandene Upload von RTF-Dateien verbessert. Die neue Upload-Option für XLSX-Dateien bildet die Grundlage zur Nutzung von Tabellenkalkulationsdokumenten in QDAcity. Dazu war neben dem Upload auch eine Unterstützung dieses Dokumenttyps innerhalb der Anwendung notwendig, weil für diese, ähnlich wie für die bereits unterstützten Textdokumente, kollaborative Bearbeitung und Codierung ermöglicht wurde. QDAcity ist dadurch besser für Projekte mit unterschiedlichen Datentypen geeignet, wobei im Sinne der Datentriangulation alle Dokumentarten unabhängig vom ihrem Datentyp mit einem gemeinsamen Codebuch codiert werden können.

2 Anforderungen

Um die Vielseitigkeit und Kompatibilität der Anwendung QDAcity zu vergrößern, sollen neue Datentypen integriert werden. Die der Umsetzung zugrunde liegenden Anforderungen sind auf die zwei Kapitel funktionale und nicht-funktionale Anforderungen aufgeteilt. Die funktionalen Anforderungen spezifizieren, wie sich das System verhalten soll, und die nicht-funktionalen Anforderungen beschreiben die Art und Weise, in der die funktionale Anforderungen umgesetzt werden sollen (Post, 2021).

2.1 Funktionale Anforderungen

Für eine strukturierte und einheitliche Formulierung der funktionalen Anforderungen habe ich mich für die Satzschablone für funktionalen Anforderungen von den SOPHISTen (2024) entschieden, die sich schematisch wie folgt darstellt:

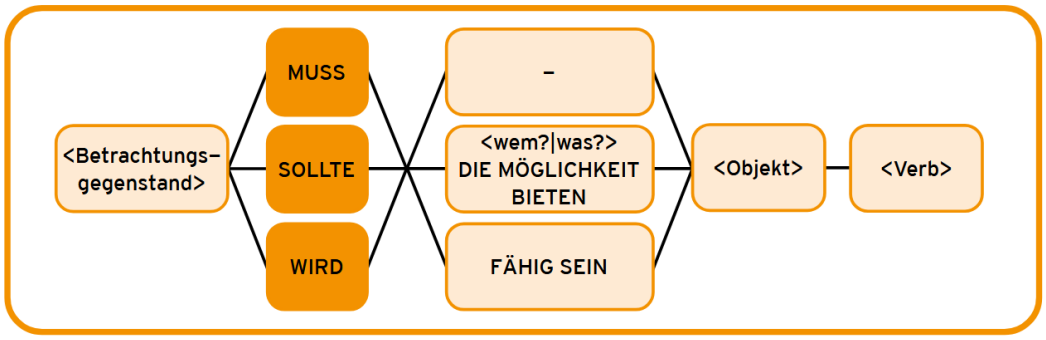


Abbildung 2.1: Satzschablone der SOPHISTen (2024) für funktionale Anforderungen

Entsprechend diesem Schema startet eine funktionale Anforderung immer mit dem *Betrachtungsgegenstand*, gefolgt von der Wichtigkeit der Anforderung. Diese wird durch die Schlüsselwörter *muss*, *sollte* und *wird* angegeben. *Muss*-Anforderungen sind verpflichtend umzusetzen, während Formulierungen mit *sollte* nur

den Wunsch eines Stakeholders darstellen. *Wird* drückt die Absicht von Stakeholdern aus, die formulierte Anforderung als Vorbereitung in einer zukünftigen Funktion zu verwenden. Deshalb ist die Realisierung einer *Wird*-Anforderungen ebenfalls verpflichtend, obwohl sie zunächst nicht getestet wird. Nach der Wichtigkeit wird die Art der Funktionalität spezifiziert. Dazu werden selbstständige Systemaktivitäten ohne Schlüsselwort, Benutzerinteraktionen mit dem Akteur und den Schlüsselwörtern *die Möglichkeit bieten* und Schnittstellenanforderungen mit den Schlüsselwörtern *fähig sein* konkretisiert. Laut Satzschablone werden die Anforderungen mit einem Objekt und einem Verb abgeschlossen (SOPHISTen, 2024).
Neben der einheitlichen Formulierung anhand der Satzschablonen wurde eine nummerische Gliederung, die die Anforderungen nach Thematik unterteilt, genutzt, um die Anforderungen zu strukturieren. Die oberste Hierarchieebene ordnet die funktionalen Anforderungen den Zielen der Erweiterung den beiden Datentypen für Textdokumente und den für die Tabellenkalkulationsdokumente zu.

2.1.1 Erweiterung der Datentypen für Textdokumente

Im Rahmen dieser Masterarbeit soll zum einen ermöglicht werden, DOCX-Dateien einem Projekt in QDAcity hinzuzufügen und diese im Editor bearbeiten zu können. Zum anderen soll der vorhandene Upload-Prozess für RTF-Dateien verbessert werden.

Erweiterung um den Datentyp DOCX

Um den neuen Datentyp DOCX einzuführen, müssen die folgenden funktionalen Anforderungen umgesetzt werden:

FA 1.1.1: Der Upload-Prozess muss dem Nutzer die Möglichkeit bieten, eine DOCX-Datei von seinem lokalen Dateisystem auszuwählen und in den Texteditor hochzuladen.

FA 1.1.2: Der Texteditor muss fähig sein, die folgenden Zeichenformatierungen, der importierten DOCX-Datei zu erkennen und als solche zu visualisieren:

FA 1.1.2.1: Der Texteditor muss fähig sein, die Formatierung *unterstrichen* bei importierten Zeichen beizubehalten.

FA 1.1.2.2: Der Texteditor muss fähig sein, die Formatierung *fett* bei importierten Zeichen beizubehalten.

FA 1.1.2.3: Der Texteditor muss fähig sein, die Formatierung *kursiv* bei importierten Zeichen beizubehalten.

FA 1.1.2.4: Der Texteditor muss fähig sein, die Formatierung *hochgestellt* bei importierten Zeichen beizubehalten.

FA 1.1.2.5: Der Texteditor muss fähig sein, die Formatierung *tiefgestellt* bei importierten Zeichen beizubehalten.

FA 1.1.2.6: Der Texteditor muss fähig sein, die Formatierung *durchgestrichen* bei importierten Zeichen beizubehalten.

FA 1.1.3: Der Texteditor muss fähig sein, die folgenden Strukturformatierungen, der importierten DOCX-Datei zu erkennen und als solche zu visualisieren:

FA 1.1.3.1: Der Texteditor muss fähig sein, die Struktur importierter Listen beizubehalten.

FA 1.1.3.2: Der Texteditor muss fähig sein, die Struktur importierter Tabellen beizubehalten.

FA 1.1.3.3: Der Texteditor muss fähig sein, die Hierarchie importierter Überschriftebenen beizubehalten.

FA 1.1.3.4: Der Texteditor muss fähig sein, die Hierarchie importierter Titel beizubehalten.

FA 1.1.4: Der Texteditor muss fähig sein, in der DOCX-Datei eingebettete Links als solche zu visualisieren.

FA 1.1.5: Der Texteditor muss fähig sein, Bilder aus einer importierten DOCX-Datei an der gleichen Stelle im Textfluss eingebunden anzuzeigen.

FA 1.1.6: Im Dokument eingebettete Bilder sollten codiert werden können.

Verbesserungen für den Datentyp RTF

Der Datentyp RTF wird bereits im Texteditor unterstützt. Die Integration dieses Datentyps soll aber im Rahmen der Arbeit verbessert werden. Dafür wurden die folgenden beiden Anforderungen definiert:

FA 1.2.1 Die für den Upload der RTF-Dateien genutzte Bibliothek *Tika* muss auf die neue Version 3.2.3 aktualisiert werden.

FA 1.2.2 Die *Latent Styles* der RTF-Dateien sollten im Texteditor nicht als sichtbarer Text erscheinen.

2.1.2 Erweiterung der Datentypen für Tabellenkalkulationsdokumente

Neben der Erweiterung der Datentypen für Textdokumente soll mit Tabellenkalkulationsdokumenten eine neue Dokumentenart in QDAcity unterstützt werden. Dazu wird neben einem neuen Tabelleneditor zum Codieren von Tabellen ein

neuer Codingtyp benötigt. Außerdem sollen XLSX-Dateien als Tabellenkalkulationsdokument-Datentyp hochgeladen werden können. Dementsprechend werden die funktionalen Anforderungen auf die drei Bereiche Tabelleneditor, Erweiterungen um den Datentyp XLSX und Codieren von Tabellen aufgeteilt.

Anforderungen an den Tabelleneditor

FA 2.1.1: Die Anwendung muss dem Nutzer die Möglichkeit bieten, ein neues, leeres Tabellenkalkulationsdokument anzulegen.

FA 2.1.2: Der Tabelleneditor sollte fähig sein, Tabellenkalkulationsdokumente, die mehrere Tabellenblätter enthalten, zu unterstützen, was durch die folgenden Anforderungen sichergestellt wird:

> **FA 2.1.2.1:** Eine Navigation muss dem Nutzer die Möglichkeit bieten, zwischen den Tabellenblättern zu wechseln.
>
> **FA 2.1.2.2:** Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, ein neues Tabellenblatt in einem Tabellenkalkulationsdokument hinzuzufügen.
>
> **FA 2.1.2.3:** Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, ein einzelnes Tabellenblatt aus einem Tabellenkalkulationsdokument zu löschen.
>
> **FA 2.1.2.4:** Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, den Namen der Tabellenblätter zu ändern.

FA 2.1.3: Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, einer Tabelle sowohl eine als auch mehrere neue Zeilen und Spalten an der vom Nutzer ausgewählten Stelle hinzuzufügen.

FA 2.1.4: Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, von ihm ausgewählte Zeilen und Spalten zu löschen.

FA 2.1.5: Die Tabellen sollten eine Zeilen- und Spaltenüberschrift haben.

FA 2.1.6: Der Tabelleneditor sollte fähig sein, Formeln innerhalb von Zellen auszuwerten.

FA 2.1.7: Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, durch Synchronisation für mehrere Nutzer, kollaboratives Arbeiten zu ermöglichen.

FA 2.1.8: Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, den Text innerhalb einer Zelle umbrechen zu lassen und manuelle Zeilenumbrüche einzufügen, damit auch längerer Text in einer Zelle angezeigt werden kann.

FA 2.1.9: Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, die Breite einer Spalte zu ändern.

Anforderungen für die Erweiterung um den Datentyp XLSX

FA 2.2.1: Der Upload-Prozess muss dem Nutzer die Möglichkeit bieten, eine XLSX-Datei auszuwählen und in den Tabelleneditor hochzuladen.

FA 2.2.2: Der Tabelleneditor muss fähig sein, die tabellarische Struktur mit den Zellwerten aus einem als XLSX-Datei hochgeladenen Tabellenkalkulationsdokument zu übernehmen.

Anforderungen für das Codieren von Tabellen

FA 2.3.1: Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, ausgewählte Zellen in einer Tabelle zu codieren.

FA 2.3.2: Der Coding-Editor muss fähig sein, gesetzte Codings der geöffneten Tabelle auf den beiden folgenden Arten anzuzeigen:

> **FA 2.3.2.1:** Der Coding-Editor muss fähig sein, gesetzte Codings in *Coding-Brackets* links neben der Datei anzuzeigen.
>
> **FA 2.3.2.2:** Der Coding-Editor muss fähig sein, den Bereich eines gesetzten Codings innerhalb der geöffneten Tabelle durch eine Umrahmung anzuzeigen.

FA 2.3.3: Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, ausgewählte Zellen, wenn sie Bestandteil eines Codings sind, aus diesem Coding zu entfernen. Wenn alle Zellen eines Codings ausgewählt sind, muss dieses Coding gelöscht werden.

FA 2.3.4: Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, das *Coding-Bracket* eines gesetzten Codings anzuklicken, um dessen codierte Zellen in der Tabelle visuell hervorzuheben.

FA 2.3.5: Der Coding-Editor muss fähig sein, beim Hinzufügen von Spalten und Zeilen innerhalb eines codierten Bereichs, dieses Coding entsprechend um die hinzugefügten Zellen zu erweitern.

FA 2.3.6: Der Coding-Editor muss fähig sein, beim Löschen von Spalten und Zeilen innerhalb eines Codings, dieses Coding entsprechend um die entfernten Zellen zu reduzieren.

FA 2.3.7 Das System sollte fähig sein, Tabellenkalkulationsdokumente bei der Bestimmung des *Intercoder Agreements* zu berücksichtigen.

FA 2.3.8: Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, die mit einem bestimmten Code codierten Zellen in einer Liste der Codings diesen Codes zu sehen.

FA 2.3.9: Der Coding-Editor muss fähig sein, Überschneidungen zweier Codings innerhalb eines Tabellenblattes in der Statistik zu berücksichtigen und diese zu visualisieren.

2.2 Nicht-funktionale Anforderungen

In diesem Kapitel wird beleuchtet, auf welcher Art und Weise diese funktionalen Anforderungen umgesetzt werden sollen. Dafür wird auf die Satzschablone der SOPHISTen (2024) für nicht-funktionale Anforderungen zurückgegriffen:

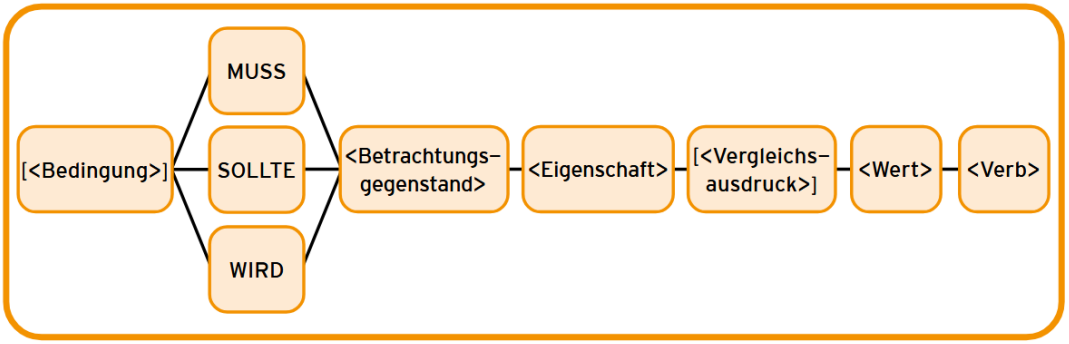


Abbildung 2.2: Satzschablone der SOPHISTen (2024) für nicht-funktionale Anforderungen

Analog zu den funktionalen Anforderungen wird auch hier die Wichtigkeit der Anforderung durch die Schlüsselwörter *muss*, *sollte* und *wird* ausgedrückt. Außerdem wird immer eine *Eigenschaft* des *Betrachtungsgegenstandes* mit einem *Wert* verglichen. Die Strukturierung der nicht-funktionalen Anforderungen erfolgt anhand des ISO-Standards 25010, der üblicherweise für eine Einschätzung der Softwarequalität herangezogen wird. Die Norm unterteilt die Eigenschaften einer Software auf die acht Bereiche funktionale Eignung, Performanz und Effizienz, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit und Übertragbarkeit aufteilt. Da funktionale Eignung funktionale Aspekte behandelt, werden die nicht-funktionalen Anforderungen den restlichen sieben Bereichen zugeordnet (Post, 2021).

2.2.1 Performanz und Effizienz

NFA 1.1: Der Inhalt der DOCX-Datei muss im Frontend in einen Blob umgewandelt und in den Google Cloud Storage (GCS) hochgeladen werden, ohne ihn an das Backend zu schicken.

NFA 1.2: Der Inhalt der RTF-Datei sollte im Frontend in einen Blob umgewandelt und in den GCS hochgeladen werden, ohne ihn an das Backend zu schicken.

NFA 1.3: Der Inhalt der XLSX-Datei muss im Frontend in einen Blob umgewandelt und in den GCS hochgeladen werden, ohne ihn an das Backend zu schicken.

NFA 1.4: Ein mit der DOCX-Datei hochgeladenes Bild muss im Frontend gecacht werden, um den Datenverkehr zum GCS gering zu halten.

2.2.2 Kompatibilität

NFA 2.1: Die Implementierung sollte für eine einheitliche Architektur auf die bereits verwendeten Technologien zurückgreifen.

2.2.3 Benutzbarkeit

NFA 3.1: Die Benutzeroberfläche muss die Sprachen Deutsch und Englisch unterstützen.

NFA 3.2: Die Benutzeroberfläche sollte die Web Content Accessiblity Guidelines (WCAG) 2.1 der Stufe AA erfüllen.

NFA 3.3: Die Benutzeroberfläche müssen dem QDAcity Designsystem entsprechen

NFA 3.4: Bei der Implementierung von Buttons sollte auf wiederverwendbare Komponenten zurückgegriffen werden.

2.2.4 Zuverlässigkeit

NFA 4.1: Der Rollout einer neuen Funktion muss über ein Feature-Flag im lokalen Speicher des Browsers für den normalen Nutzer verborgen sein, bis die Funktionalität für alle freigegeben ist.

NFA 4.2: Es muss sichergestellt werden, dass bereits vorhandene Funktionalität durch Änderungen nicht gestört wird.

2.2.5 Sicherheit

NFA 5.1: Nur Nutzer mit Leseberechtigung für die Datei dürfen in der Lage sein, ein Bild, das beim Upload-Prozess einer DOCX-Datei in den GCS hochgeladenen wird, herunterzuladen.

NFA 5.2: Beim Upload-Prozess einer DOCX-Datei müssen Cross-Site-Scripting (XSS)-Attacken verhindert werden.

NFA 5.3: Bei allen Requests an das Backend muss der Nutzer authentifiziert werden.

NFA 5.4: Die Daten müssen verschlüsselt in der Datenbank gespeichert werden.

NFA 5.5: Die in QDAcity verwendete rollen-basierte Zugriffskontrolle sollte verwendet werden, um bei Projektteilnehmern zwischen lesendem und schreibendem Zugriff unterscheiden zu können.

2.2.6 Wartbarkeit

NFA 6.1: Um die HTML-zu-Slate-Node-Umwandlung aus dem Collaborative-Editing-Service (CES) im Frontend wiederzuverwenden, dürfen die benötigten Methoden manuell kopiert werden. Für eine einfachere Wartung sollten die Kopien der Methoden im Frontend und die Originale im CES identisch und das Vorgehen dokumentiert sein.

NFA 6.2: Alle neuen Endpunkte im Backend sollten eine Codezeilentestabdeckung mit Unit-Tests von 100% haben.[1]

NFA 6.3: Alle neuen Funktionalitäten müssen mit Akzeptanztests geprüft werden.

NFA 6.4: Alle Java-Methoden, deren Umfang über eine einzeilige Implementierung hinausgeht, sollten mit *JavaDoc*-Kommentaren dokumentiert werden.

NFA 6.5: Das Prinzip der Trennung von Verantwortlichkeiten sollte eingehalten werden.

[1] https://gitlab.com/qdacitygroup/qdacity/-/wikis/How-to-write-unit-tests

3 Architektur

3.1 Ausgangslage

In diesem Kapitel wird die für die Umsetzung der Anforderungen relevante Architektur vorgestellt, auf die in den folgenden Kapiteln dargestellten eigenen Arbeiten aufbauen. Der Fokus liegt deshalb auf dem CES mit der Synchronisation für das kollaborative Arbeiten, der Verarbeitung von Textdokumenten im Allgemeinen und deren Codierung.

3.1.1 Das CES-Modul als *Hocuspocus-Server* und die Kommunikation mit dem Frontend

Neben einem Frontend mit React-Framework und einem Java-basierendem Backend existiert im QDAcity-Projekt auch der CES. Dieser Service ermöglicht ein kollaboratives Arbeiten an Textdokumenten unter Verwendung von Yjs[1] YDocs. *Yjs* nutzt die Eigenschaften von CRDTs, um Änderungen an YDocs durch *Update*-Nachrichten konfliktfrei zu synchronisieren. Außerdem fungiert der Service als Schnittstelle zwischen Frontend und GCS und ist für das Speichern und Herunterladen von den YDocs zuständig. Dazu ist der Service als *Hocuspocus-Server*[2] konfiguriert, was ermöglicht, dass die *Update*-Nachrichten der YDocs synchronisiert werden, sodass mehrere Benutzer parallel am gleichen Dokument arbeiten können. Der *Hocuspocus-Server* wurde um eine *Custom-Extension*[3] erweitert, die den Zugriff auf die Datenbank kontrolliert. Da diese Erweiterung teilweise vom Dokumenttyp abhängig ist, gibt es für jeden Dokumenttypen einen eigenen *DocHandler*, der die benötigten *Lifecycle-Hooks*[4] *onLoadDocument*, *onAuthenticate* und *onStoreDocument* für den jeweiligen Dokumenttyp bereitstellt.
Im Fontend wurde der *Hook useHocuspocusProvider* implementiert, der als *Hocuspocus-Provider* dient. Dieser *Hook* stellt die Verbindung zum *Hocuspocus-Server*

[1] https://docs.yjs.dev/
[2] https://tiptap.dev/docs/hocuspocus/server/configuration
[3] https://tiptap.dev/docs/hocuspocus/guides/custom-extensions
[4] https://tiptap.dev/docs/hocuspocus/server/hooks

her und gibt das YDoc zurück. Mit einem Aufruf dieses *Hook* kann der Coding-Editor auf das YDoc zugreifen, wobei durch die *Hocuspocus*-Verbindung sichergestellt ist, dass das YDoc in Echtzeit synchronisiert wird.

3.1.2 Textdokumente in QDAcity

Im Texteditor konnte der Nutzer schon vor Implementierung der neuen Funktionen Textdokumente in Form von TXT-Dateien und RTF-Dateien aus seinem lokalen Speicher in ein Projekt hochladen. Dafür ist im Backend die Unterklasse *TextDocument* vorhanden, die neben den Unterklassen *PDFDocument* und *TranscriptionDocument* von der abstrakten Klasse *BaseDocument* erbt. Diese Hierarchie ist im Klassendiagramm in der Abbildung 3.1 zu sehen.

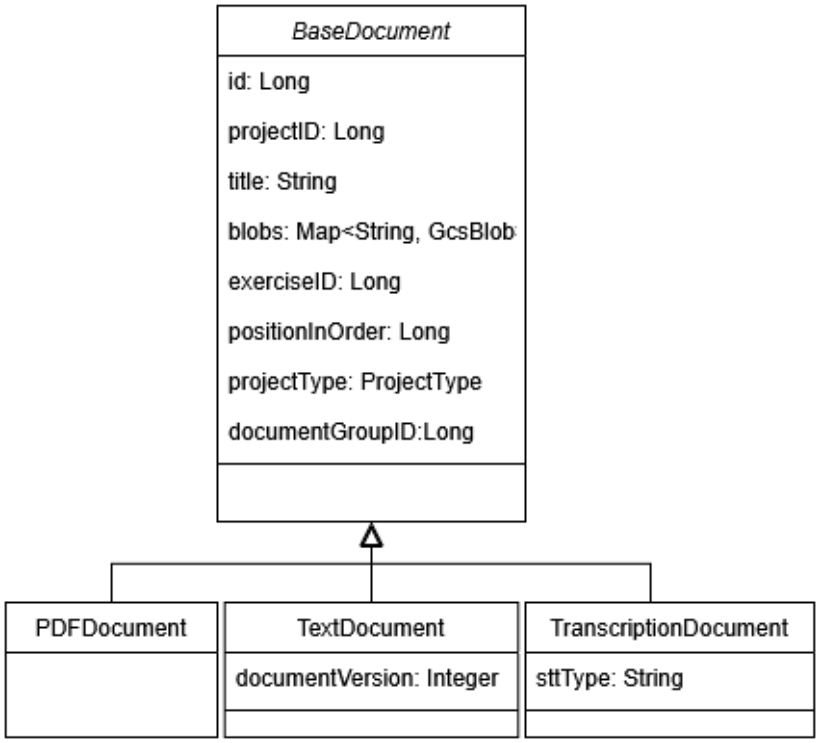


Abbildung 3.1: Klassendiagramm der Dokumente

Diese Klassen enthalten neben den Metadaten des Dokuments auch die Map *blobs* als Attribut. Diese Map verknüpft das Dokument mit den *Blobs* des GCS, die den Inhalt der Dokumente beinhalten. Den Textdokumenten sind jeweils die Blobs *yDoc* und *content* zugeordnet. Zusätzlich zu den allgemeinen Metadaten wird die *documentVersion* persistiert, die angibt, ob das Dokument bereits in einem YDoc synchronisiert wird. Beim Upload einer TXT-Datei wird der Inhalt initial im *content*-Blob gespeichert. Dabei wird die *documentVersion* auf 1 gesetzt, was signalisiert, dass noch kein YDoc für dieses Dokument existiert. Bei

RTF-Dateien wird analog vorgegangen, aber der Inhalt wird im Backend von RTF zu HTML umgewandelt, bevor dieser in den *content*-Blob gespeichert wird. Für diese Konvertierung wird die Java-Bibliothek *Apache Tika*[5] verwendet. Anhand der *documentVersion* erkennt der CES, dass das Dokument noch für ein kollaboratives Arbeiten als YDoc persistiert werden muss. Innerhalb eines YDocs wird der Inhalt der Datei als Slate-Nodes[6] persistiert. Die Kombination dieser beiden Technologien wird durch die Middleware *slate-yjs*[7] vereinfacht. Dazu ist in der *SlateUtils*-Datei ein Algorithmus implementiert, der die HTML-Struktur in eine Slate-Nodes-Hierarchie umwandelt. Die in HTML umgeformten Slate-Nodes werden dann in einem YDoc unter dem Schlüssel *content* als *XmlText* persistiert und die *documentVersion* auf 2 gesetzt, um zu signalisieren, dass beim nächsten Herunterladen direkt auf das YDoc zugegriffen werden kann.

3.1.3 Codieren der Dokumente

Für das Codieren von den unterschiedlichen Dokumentarten gibt es verschiedene Coding-Typen, die in der Abbildung 3.2 zu sehen sind. Alle Klassen erben von der abstrakten Klasse *BaseCoding* die Attribute der Metadaten des Codings. Die Attribute zur Position des Codings innerhalb des Dokuments unterscheiden sich je nach Art des Codings. Auf der ersten Ebene wird zwischen Codieren eines Bereichs oder eines Textes differenziert. Das Codieren eines Bereichs wird bisher nur für PDF-Dokumente mit der Klasse *PDFAreaCoding* unterstützt. Das *TextCoding* hat zwei Unterklassen, einerseits das *PDFTextCoding* für PDF-Dokumente und anderseits *SlateTextCoding* für Textdokumente, die im YDoc als Slate-Nodes gespeichert werden. Damit die Codings zwischen mehreren Nutzern, die parallel am gleichen Textdokument arbeiten, synchronisiert werden können, werden diese im YDoc des Textdokuments gespeichert. Neben dem Inhalt, der unter dem Schlüsselbegriff *content* persistiert ist, werden die Codings in einer Map unter dem Schlüssel *codings* gespeichert. Der CES synchronisiert die Codings, die über die Hocuspocus-Verbindung in die Frontends mehrerer Nutzer geladen werden können.

[5] https://tika.apache.org/
[6] https://docs.slatejs.org/
[7] https://github.com/BitPhinix/slate-yjs

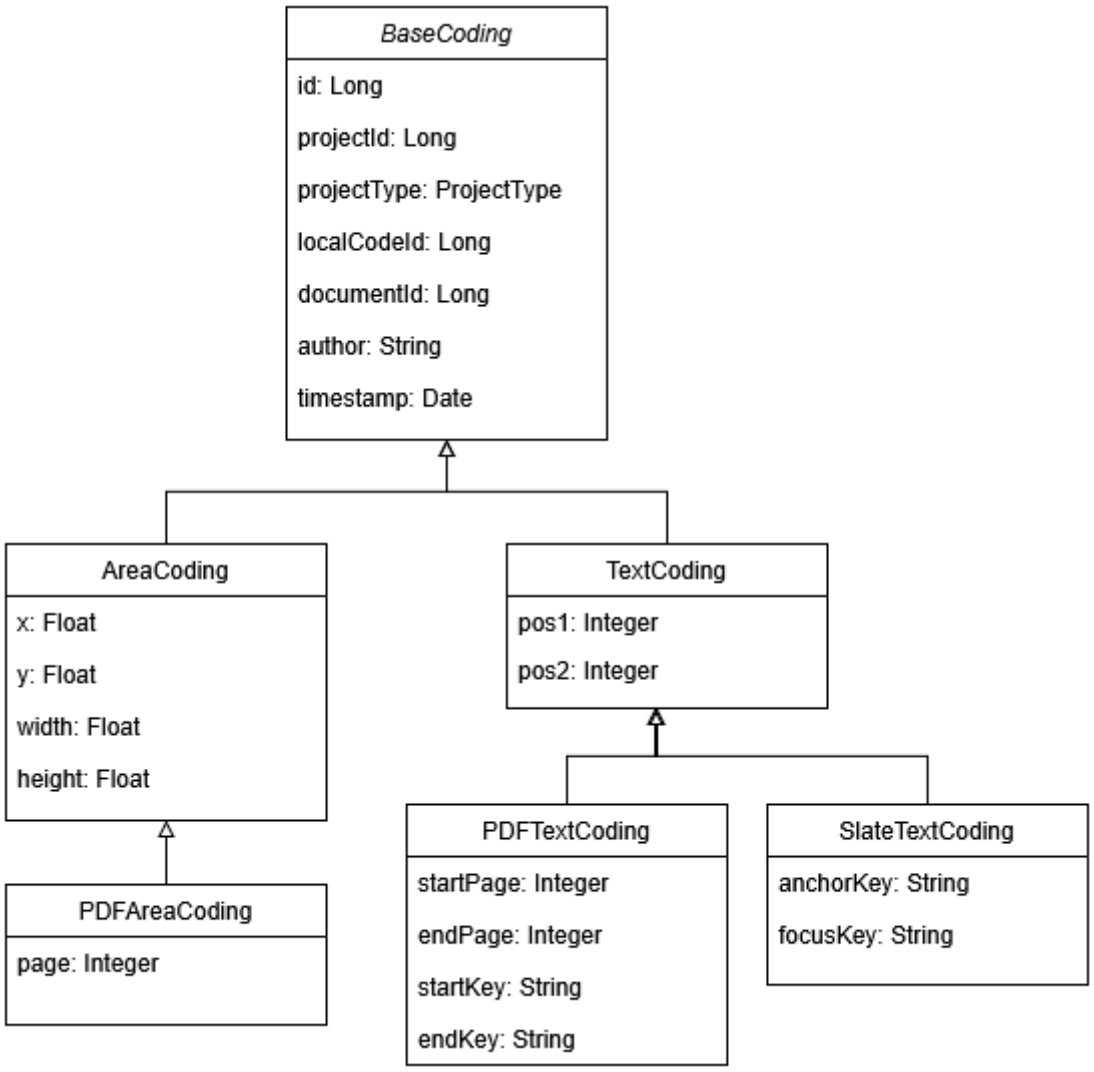


Abbildung 3.2: Klassendiagramm der Coding-Klassen

3.2 Konvertierung der DOCX-Dateien

Um die Unterstützung des Dateiformats DOCX zu ermöglichen, wird eine neue Konvertierung zu den vom QDAcity-Texteditor verwendeten und für kollaborative Arbeiten notwendigen Slate-Nodes benötigt. Für diese Umwandlung wurde ein Umweg über HTML gewählt, damit die bereits im Projekt vorhandene Umwandlung von HTML zu Slate-Nodes beim Hochladen von RTF-Dateien weiter verwendet werden kann.

3.2.1 Evaluierung von Bibliotheken zur Konvertierung von DOCX zu HTML

Um die Konvertierung von DOCX zu HTML effizient zu implementieren, sollte auf vorhanden Bibliotheken zurückgegriffen werden. Die in Betracht gezogenen Bibliotheken, die eine solche Umwandlung ermöglichen, werden im folgenden verglichen. Dazu werden die Eigenschaften der einzelnen Bibliotheken tabellarische gegenübergestellt. Anschließend wird die Entscheidung für die Bibliothek Mammoth[8] begründet.

Tabelle 3.1: Gegenüberstellung verschiedener Bibliotheken zur Konvertierung von DOCX zu HTML

Bibliothek	Mammoth[9]	Docx4j[10]	Pandoc[11]	officeParser[12]
Lizenz	BSD-2 Clause	Apache	GPL-2.0	MIT
Kompatibilität	DOCX	DOCX, XLSX	DOCX, RTF, ODT, CSV	DOCX, XLSX, ODT
Stars auf Github	5.8k	2.3k	39.4k	236
Datum des letzten Releases	2026[13]	2019[14]	2026[15]	2026[16]
Größe der Bibliothek	2,16 MB[17]	5.2 MB[18]	30-40 MB[19]	57.9kB [20]
Laufzeitumgebung	Frontend (Javascript)	Backend (Java)	Als eigenständiger Service aufgrund der Lizenzbestimmungen	Frontend (Javascript)
Konvertierung Layouts	Fokus auf semantische Informationen	Möglichst genaue Kopie	Komplexes HTML mit nicht immer gleicher Struktur	Keine Konvertierung des Layouts

[8]https://github.com/mwilliamson/mammoth.js

Nach gründlicher Recherche habe ich mich für die Javascript Bibliothek *Mammoth* entschieden. Im Sinne einer transparenten Entscheidungsfindung werden im folgenden die Vorteile dieser Bibliothek gegenüber ihren Alternativen bewertet. Die Bibliothek *OfficeParser* wurde ausgeschlossen, weil sie im Vergleich zu *Mammoth* deutlich weniger bekannt ist, was man unter anderem an der Anzahl der Github Stars erkennen kann. Durch die geringe Bekanntheit bestehen die Risiken, dass die Bibliothek nicht ausreichend erprobt ist und keine regelmäßige Updates bekommt.
Die Benutzung der Bibliothek *Pandoc* wird durch die Lizenzbestimmungen der GPLv2-Lizenz erschwert. Diese schreibt vor, dass eine Software, die GPLv2-lizenzierten Code einbindet und davon ein abgeleitetes Werk ist, welches verbreitet wird, mit derselben Lizenz veröffentlicht werden muss (Jaeger, 2011). Da QDAcity nicht unter einer GPL-kompatiblen Lizenz steht, kann die Bibliothek *Pandoc* nicht direkt eingebunden werden. Deshalb würde nur die Option bleiben, *Pandoc* als einen separaten Service in der Google Cloud Plattform (GCP) zu betreiben, auf den die QDAcity-Anwendung über eine Schnittstelle zugreift. In diesem Fall zählt QDAcity üblicherweise nicht als *derivative work* und die Pflicht zur Lizenzierung unter GPLv2 entfällt. Doch eine solche Konstellation führt zu erheblichem Mehraufwand und zusätzlichen Kosten für den laufenden *Pandoc-Service* in der GCP, die nicht im Verhältnis zu dem Nutzen der Bibliothek stehen. *Mammoth* steht hingegen unter der BSD-2-Clause Lizenz, die keine lizenzrechtlichen Vorgaben für das Einbinden und Veränderungen der Bibliothek macht, solange der Urhebervermerk, die Lizenzbestimmungen sowie der Haftungs- und Gewährleistungsausschluss ebenfalls mitverbreitet werden (Jaeger, 2011).
Docx4j hat den großen Vorteil, dass nicht nur DOCX-Dateien unterstützt werden, sondern auch XLSX-Dateien, die ebenfalls im Rahmen der Masterarbeit benötigt werden. Doch diesem Vorteil stehen zwei gravierenden Nachteile gegenüber. *Docx4j* ist eine Java-Bibliothek und muss deshalb im Backend laufen. Da aber das Backend von QDAcity in der GCP läuft, würde die Nutzung der zusätzlich für *Docx4j* benötigten Ressourcen merkliche Zusatzkosten verursachen. Deshalb wird eine Lösung, die im Frontend und damit auf den Ressourcen des Nutzers laufen, bevorzugt (Vergleich NFA 1.1). Außerdem wird berichtet, dass

[9] https://github.com/mwilliamson/mammoth.js
[10] https://github.com/plutext/docx4j
[11] https://github.com/jgm/pandoc
[12] https://github.com/harshankur/officeParser
[13] https://www.npmjs.com/package/mammoth?activeTab=versions
[14] https://mvnrepository.com/artifact/org.docx4j/docx4j/versions
[15] https://github.com/jgm/pandoc/releases
[16] https://www.npmjs.com/package/officeparser?activeTab=versions
[17] https://www.npmjs.com/package/mammoth
[18] https://mvnrepository.com/artifact/org.docx4j/docx4j/6.1.2
[19] https://github.com/jgm/pandoc/releases/tag/3.8.2.1
[20] https://www.npmjs.com/package/officeparser

Mammoth im Vergleich zu *Docx4j* einen deutlich saubereren HTML-Code produziert, da *Mammoth* nicht versucht, die Formatierung so exakt zu erhalten wie *Docx4j* dies tut.[21]

3.2.2 Allgemeiner Prozess zur Konvertierung

Auf Basis der Entscheidung für die Bibliothek *Mammoth* zur Umwandlung der DOCX-Datei in HTML, wurde ein Prozess implementiert, der sicherstellt, dass die gespeicherte Datei nach der Konvertierung eine Dateistruktur erhält, die identisch zu der von konvertierten RTF und TXT Dateien ist. Dadurch können alle bereits vorhandenen Implementierungen für Textdateien ohne zusätzliche Anpassungen weiter verwendet werden und die Anwendung kann, unabhängig vom ursprünglichen Dateiformat, mit allen Texten umgehen. Für die Umwandlung der DOCX-Dateien soll aber auf den Umweg über Backend und Konvertierung zu Slate-Nodes im CES, wie er bei RTF-Dateien genutzt wird, verzichtet werden (vgl. NFA 1.1). Deshalb wird die Umformungslogik der Datei *SlateUtils* ins Frontend kopiert. Da allerdings das CES in einer Node.js Umgebung und das Frontend mit React-Framework im Browser läuft, muss eine Fallunterscheidung implementiert werden, die abhängig von dem Kontext einen DOM entweder über die dynamisch geladene Node.js Bibliothek *jsdom* aufbaut oder in React den Browser-DOM verwendet. Damit beide Dateien einfach in Zukunft gewartet werden können, sollten sie aber identisch sein, sodass eine Anpassung in der einen Klasse schnell in die jeweils andere Klasse übernommen werden kann (vgl. NFA 6.1). Deshalb wurde in beiden Dateien eine Fallunterscheidung eingebaut, die dynamisch bei Laufzeit abhängig vom mitgegebenen Kontext entscheidet, welcher DOM verwendet werden soll. In der *SlateUtils* Datei existiert die Deserialisierungsfunktion, die rekursiv die einzelnen DOM Elemente zu Slate-Knoten umwandelt. In *Slate.js* wird dabei zwischen Container-Element-Knoten mit semantischen Bedeutung und den Blättern als *Text-Knoten* unterschieden, die den Textinhalt sowie dessen Formatierungen enthalten[22]. Beim Deserialisieren wird über eine *switch-case*-Struktur das DOM-Element abhängig vom HTML-Tag zu einem entsprechenden Slate-Element oder Text-Knoten umgeformt. Mit diesem Prozess wird die Baumstruktur in Slate aufgebaut. Um diese in dem *GCS* zu persistieren, wird ein neues YDoc erstellt und mit einem Slate-Editor verbunden. In diesen Editor wird anschließend die konvertierte Baumstruktur geladen und somit im YDoc gespeichert. Mit der Methode *encodeStateAsUpdate(yDoc)* aus der *yjs* Bibliothek werden die Änderungen an dem YDoc als *Update*-Nachricht serialisiert.[23] Diese *Update*-Nachricht wird dann in den GCS hochgeladen. Der beschriebene Konvertierungsprozess ist in einem Sequenzdiagramm in der Abbildung 3.3 visualisiert.

[21] https://kaosktrl.wordpress.com/2016/04/28/converting-docx-to-html-using-mammoth/
[22] https://docs.slatejs.org/concepts/02-nodes
[23] https://docs.yjs.dev/api/document-updates

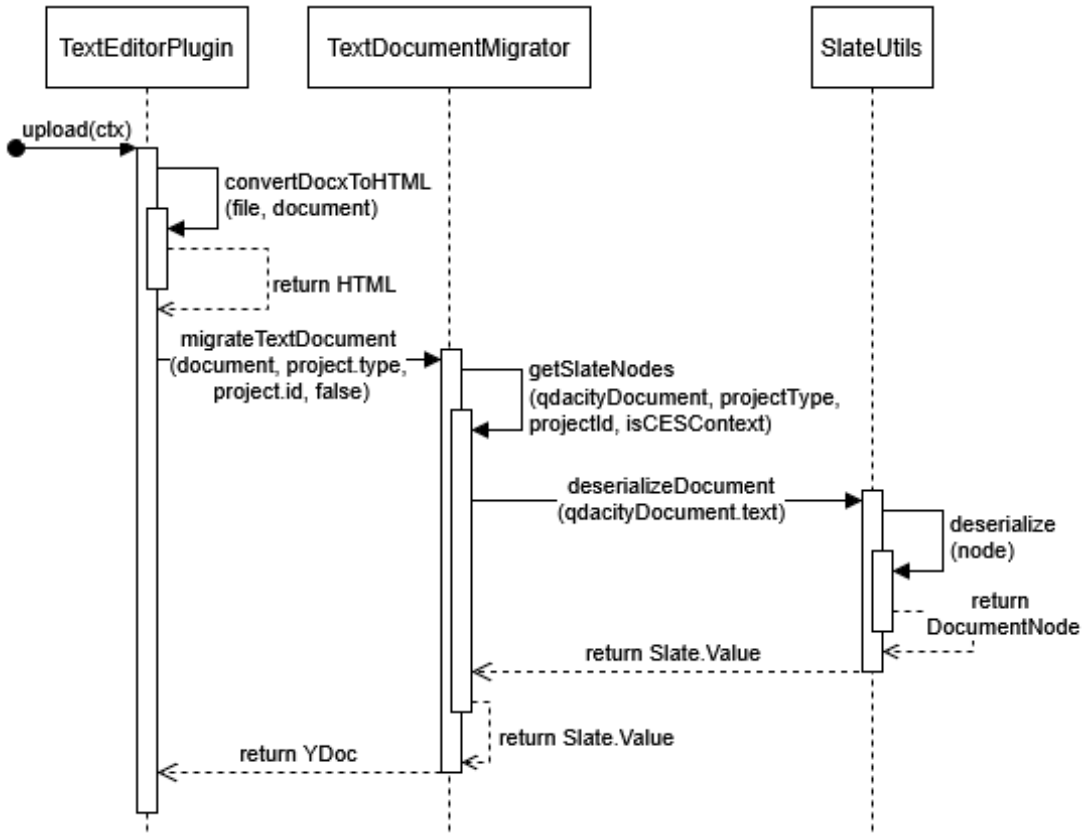


Abbildung 3.3: Sequenzdiagramm zur Umwandlung von DOCX-Dateien zu YDocs

3.2.3 Datenübertragung und Caching-Strategie

In diesem Kapitel wird erläutert, wie die Datenübertragung und das Caching der DOCX-Dateien implementiert wurde.

Datenübertragung und Caching-Strategie des YDocs

Um die *Update*-Nachricht, die aus den Slate-Nodes der hochgeladenen DOCX-Datei erzeugt wurde, direkt im Frontend in den GCS hochzuladen, wurde auf die vorhandene Funktion *uploadArray* zurückgegriffen. Diese bekommt über den Endpunkt *getSignedUploadUrlsForDocuemntId* vom Backend eine signierte Upload-URL, die einen zeitlich befristeten schreibenden Zugang auf ein bestimmtes Objekt im GCS gewährt.[24] Über diese URL wird dann die *Update*-Nachricht als Blob in den GCS hochgeladen. Da nach dieser initialen Umwandlung die ursprüngliche DOCX-Datei wie die anderen Textdateien als YDoc persistiert ist, greifen hierbei die vorhandenen Mechanismen für die Synchronisation und dem Caching im

[24] https://docs.cloud.google.com/storage/docs/access-control/signed-urls?hl=de

lokalen Speicher des Browsers.

Datenübertragung und Caching-Strategie der über DOCX-Dateien hochgeladenen Bilder

Neben dem in dem GCS hochgeladenem YDoc wird für jede hochgeladene DOCX-Datei im Backend ein Objekt der Klasse *TextDocument* erzeugt und persistiert, welches die Metadaten des Dokuments enthält. Diese Klasse, die im Klassendiagramm in Abbildung 3.1 visualisiert wird, wurde um das neue Attribut *imageMap* vom Datentyp *HashMap* mit der *imageId* als Schlüssel des Typs String und dem zugehörigen *GcsBlob* als Wert erweitert. Um ein Bild als Blob in den *GCS* hochzuladen, wurde der neue Endpunkt *getSignedImageUploadUrlForTextDocumentId* implementiert, der als Parameter die Id des Dokuments und die Id des Bildes benötigt. Beim Aufruf dieses Endpunkts wird ein neuer *GcsBlob* erzeugt, der mit dem Schlüssel der mitgelieferten Id des Bildes, in der *imageMap* des zugehörigen *TextDocument*-Objekts gespeichert wird. Der Endpunkt gibt eine signierte Upload-URL für diesen neu erzeugten Blob zurück. Der genaue Ablauf kann dem Sequenzdiagramm in Abbildung 3.4 entnommen werden.

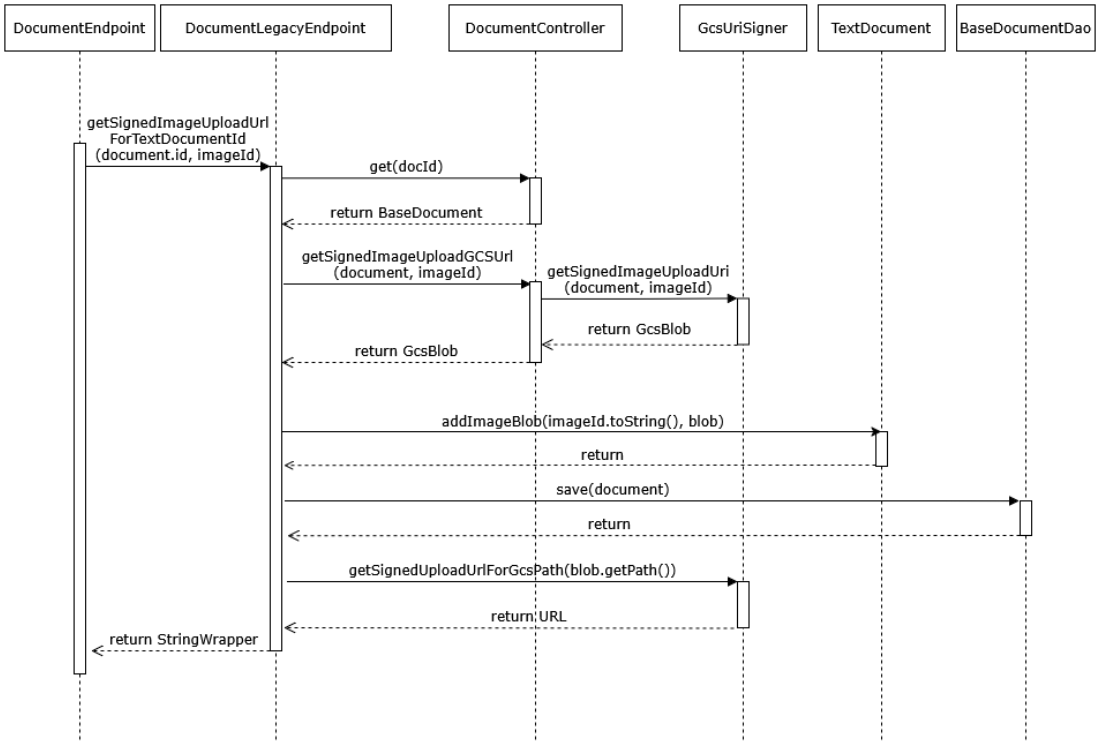


Abbildung 3.4: Sequenzdiagramm zum Upload eines Bildes

Außerdem wurde der neue Endpunkt *getSignedDownloadImageUrlsForTextDocumentId* hinzugefügt, der für alle Bilder, die in dem Dokument mit der übergebenen Id eingebetteten sind, die signierten Download-URLs zurückgibt. Dafür wird eine *HashMap* erzeugt, deren Schlüssel die einzelnen Ids der Bilder sind und die als Wert für jedes Blob eine Download-URL enthält. Diese *HashMap* wird durch das Iterieren über die *imageMap* und Erzeugen der signierten URLs für jedes gespeicherte Blob generiert. Zusätzlich wurde der neue Endpunkt *getSignedDownloadImageUrlsForTextDocumentIdForReview* implementiert. Dieser unterscheidet sich nur in der Autorisierung von dem Endpunkt *getSignedDownloadImageUrlsForTextDocumentId* und ruft danach intern die gleichen Methoden auf. Während über *getSigendDownloadImageUrlsForTextDocumentId* nur der Nutzer, der für dieses bestimmte Dokument ein Autorisierung besitzt, die Download-URLs abrufen kann, bekommt über den Endpunkt *getSigendDownloadImageUrlsForTextDocumentIdForReview* jeder, der eine Review für das Dokument durchführen darf, die Download-URLs für die Bilder.
Im Frontend wurde die Methode *uploadImageBlob* implementiert, die für jedes hochzuladende Bild aufgerufen wird und als Parameter, das Dokument, den Namen des Blobs und das Blob als File-Objekt erwartet. Nach dem Generieren einer zufälligen Id des Bildes wird in dieser Methode der neue Endpunkt *getSignedImageUploadUrlForTextDocumentId* aufgerufen und mithilfe der zurückgegebenen signierten Upload-URL das Blob mit der bereits vorhandenen Methode *uploadBlobChunked* in den *GCS* hochgeladen.
Parallel zum Upload wird das Bild auch direkt in dieser Methode gecacht, um den Datenverkehr zum *GCS* zu reduzieren (vgl. NFA 1.4). Für das Caching verwendet QDAcity den lokalen Speicher des Browsers vom Benutzer. In diesem wird, falls noch nicht vorhanden, die *IndexedDB imageStorage* mit dem *Object-Store uploads* angelegt, der als Primärschlüssel eine ID erwartet. Der hochgeladene Blob wird mit der *imageId* als Schlüssel in den *Object-Store* gespeichert.
Um die Bilder für den Texteditor herunterzuladen, wurde die Methode *downloadImageBlobs* implementiert, die das Dokument selbst sowie die zugehörige *imageMap* als Parameter erwartet. Der Ablauf dieser Methode wird in dem Aktivitätsdiagramm in Abbildung 3.5 visualisiert. Beim Aufruf wird zuerst für jeden Schlüssel der *imageMap* geprüft, ob das Bild bereits in der *IndexedDB imageStorage* zwischengespeichert ist. Wenn dies der Fall ist, können diese Blobs direkt mit den entsprechenden Ids der Bilder zurückgegeben werden. Falls nicht, wird abhängig davon, ob die Bilder für ein Review oder für ein eigenes Dokument benötigt werden, der entsprechende Endpunkt *getSignedDownloadImageUrlsForTextDocumentId* oder *getSignedDownloadImageUrlsForTextDocumentIdForReview* aufgerufen. Auch das Ergebnis dieses Aufrufs wird in der *IndexedDB qdacity-app-runtime* gecacht, um den Datenverkehr zum Backend zu reduzieren und beim nächsten Aufruf direkt darauf zuzugreifen zu können. Bei Wiederverwendung der Download-URLs aus dem Cache muss allerdings beachtet werden, dass diese nur zeitlich befristet gültig sind. Deshalb muss im Fehlerfall der Cache

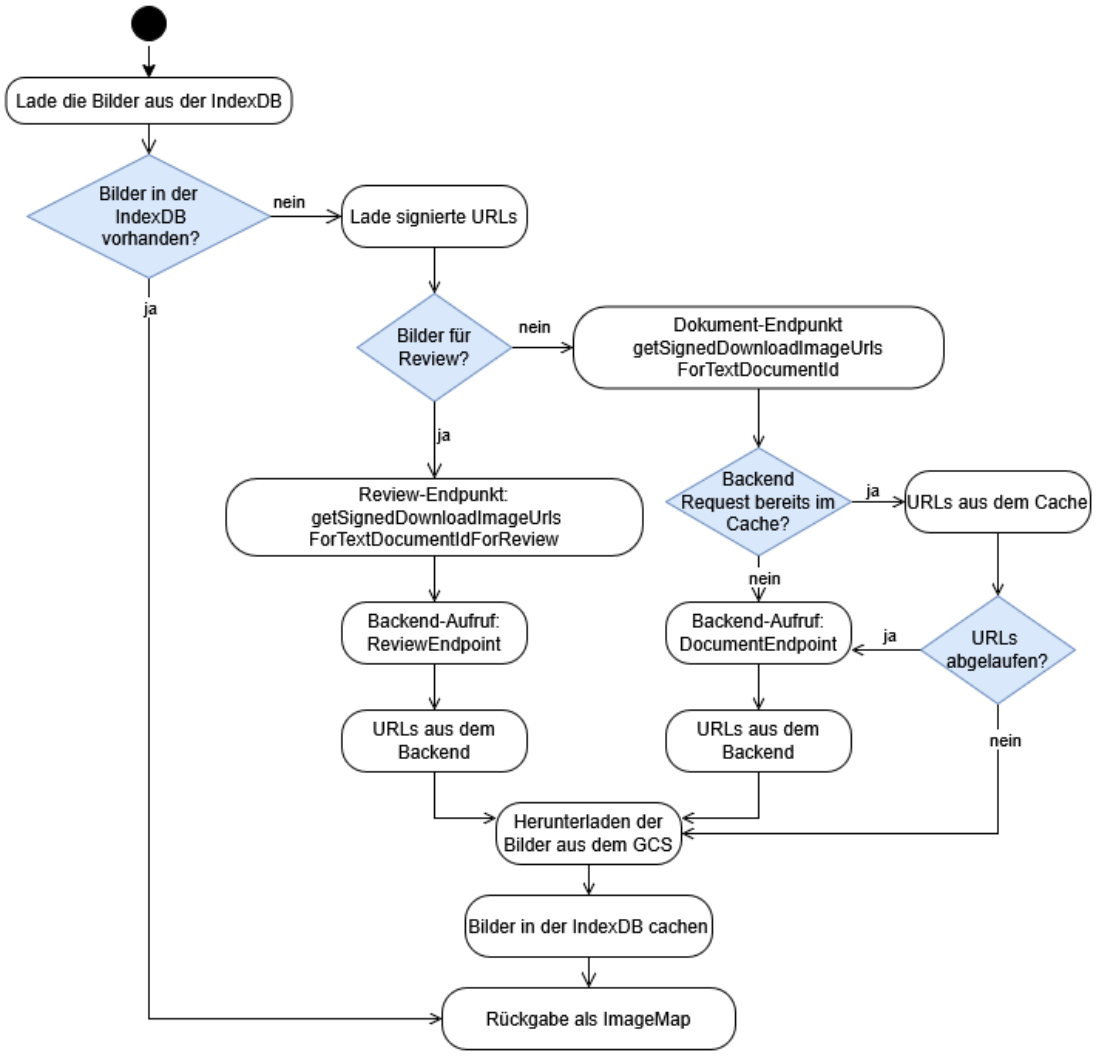


Abbildung 3.5: Aktivitätsdiagramm für *downloadImageBlobs*

umgangen werden, und es müssen neue URLs im Backend erzeugt werden. Mithilfe der Download-URLs können die Bilder dann einzeln aus dem GCS heruntergeladen und direkt im *imageStorage* zwischengespeichert werden. Der Rückgabewert der Methode ist eine Liste, die aus einer Liste für jedes Bild besteht, die das aus dem Blob erzeugte File-Objekt mit der jeweiligen Id des Bildes enthält.

3.3 Architektonische Entscheidungen für die Integration von Tabellenkalkulationsdokumenten

Um Tabellenkalkulationsdokumente in QDAcity zu unterstützen, müssen neue Strukturen aufgebaut werden, weil sich diese Dokumente grundlegend von den bereits vorhandenen Dokumentarten Text, PDF, Audio und Transkription unterscheiden und daher auch einen neuen Datentyp für Codierungen erfordert.

3.3.1 Speicherstrukturen für Tabellenkalkulationsdokumente

Für das Persistieren der Metadaten für Tabellenkalkulationsdokumente wurde die neue Klasse *SpreadsheetDocument* als Unterklasse der abstrakten Klasse *BaseDocument* aus Abbildung 3.1 erstellt, siehe Abbildung 3.6. Diese Klasse hat als

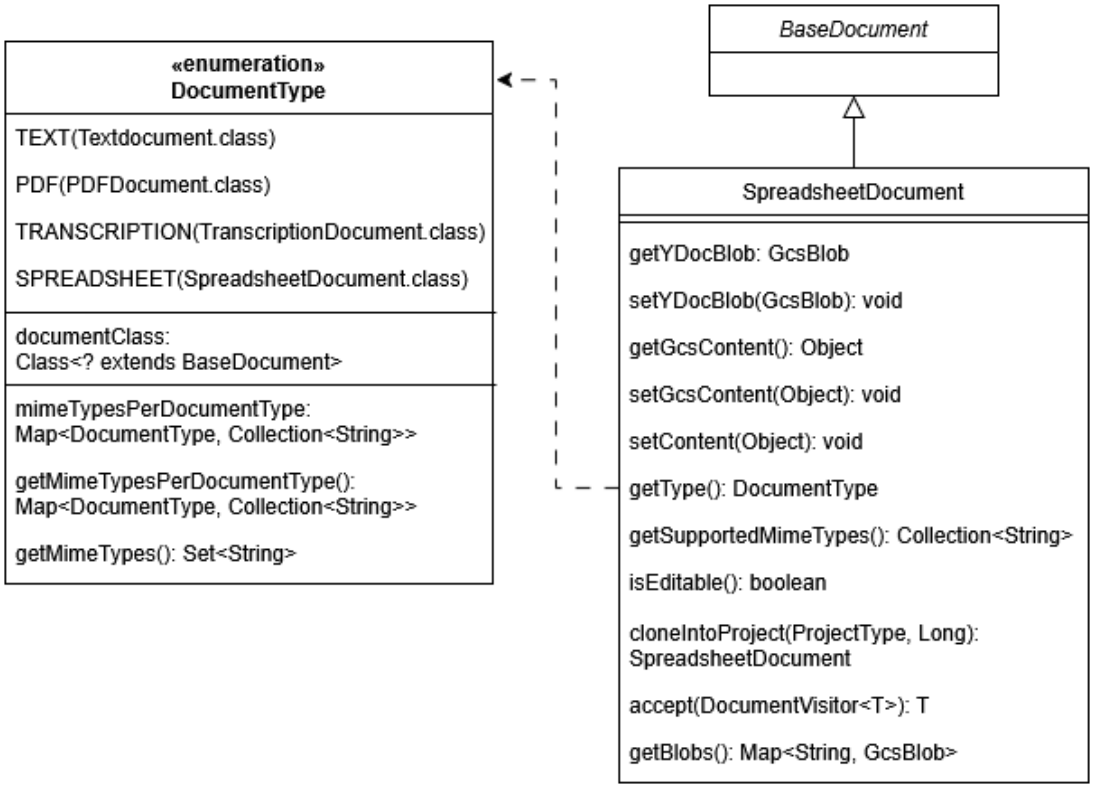


Abbildung 3.6: Klassendiagramm von der Klasse *SpreadsheetDocument*

Rückgabewert der Methode *getSupportedMimeTypes* den Mime-Typ für XLSX-Dateien, nämlich *application/vnd.openxmlformats-officedocument.spreadsheetml.*

sheet[25], da dieses Dateiformat als Tabellenkalkulationsdokument behandelt werden soll, siehe Kapitel 4.3.6. Weil das Dokument bearbeitbar sein soll, gibt die Funktion *isEditable* immer den Wert *true* zurück. Für die Rückgabe der Methode *getType* wurde das Enum *DocumentType* um den Wert *SPREADSHEET* mit der zugehörigen *documentClass SpreadsheetDocument.class* erweitert. Analog zu den Textdokumenten wird der Inhalt des Tabellenkalkulationsdokuments in einem *YDoc* gespeichert, das über die Methode *getYDocBlob* erreichbar ist.
Für die kollaborative Zusammenarbeit mit mehreren Clients bietet *YDoc* die Datentypen *Y.Map*, *Y.Array*, *Y.Text*, *Y.XmlText*, *Y.XmlElement* und *Y.XmlFragment* an.[26] Um die Elemente eines Tabellenkalkulationsdokument mit diesen Datentypen zu modellieren, wurde eine Struktur von *YArrays* und *YMaps* aufgebaut, die in der Abbildung 3.7 dargestellt ist. Innerhalb des *YDocs* wird jedes

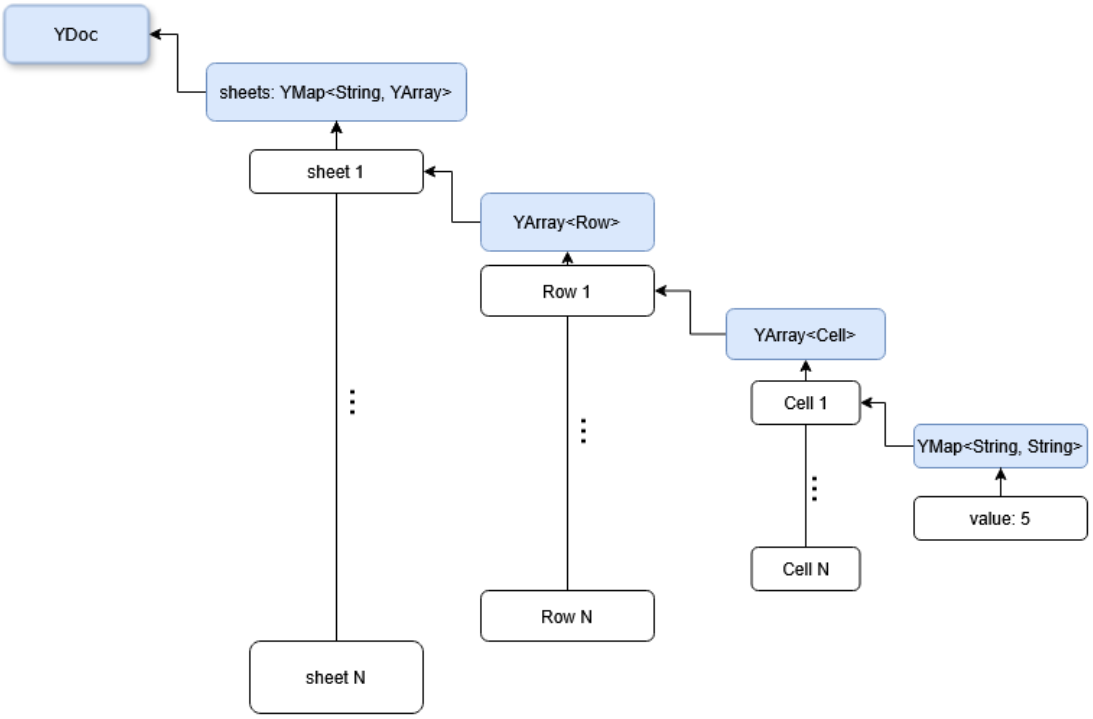


Abbildung 3.7: Speicherstruktur innerhalb des *YDocs*

Tabellenblatt als Element einer *YMap* namens *sheets* gespeichert, das als Schlüssel den Namen des Tabellenblatts hat und dessen zugehöriger Wert wieder ein *YArray* ist. Ein Element in diesem untergeordneten *YArray* entspricht eine Zeile der Tabelle, deren Wert ein weiteres *YArray* ist, das die Zellen der Zeile beinhaltet. Jede einzelne Zelle wird ebenfalls als *YMap* gespeichert. Der Wert in

[25]https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/MIME_types/Common_types
[26]https://docs.yjs.dev/api/shared-types

dieser *YMap* in der untersten Ebene ist der Wert der Tabellenzelle, der unter dem Schlüsselwort *value* gespeichert ist. Auf dieser Art und Weise werden alle Informationen des Dokuments in Datenstrukturen gespeichert, die ein kollaboratives Arbeiten unterstützen. Das YDoc wird wie die Textdokumente als Blob in dem GCS persistiert.

3.3.2 Bibliotheksvergleich zur Visualisierung der Tabellen

Um den Entwicklungsaufwand zu reduzieren, wird für die Visualisierung der Tabelle im Editor von QDAcity auf eine bereits existierende Bibliothek und damit etablierte Lösung zurückgegriffen. Ausgewählt wurde die Bibliothek *ReactGrid*[27]. Die Gründe für diese Wahl werden im folgenden anhand einer Gegenüberstellung der Eigenschaften einer Auswahl von vielversprechenden Bibliotheken dargelegt, wobei die Vor- und Nachteile gegenüber gestellt werden.

Tabelle 3.2: Gegenüberstellung verschiedener Bibliotheken zur Visualisierung von Tabellen

Bibliothek	ReactGrid[28]	react-data-grid[29]	X Data Grid[30]	handsontable[31]
Lizenz	MIT	MIT	Aufteilung in kostenlos mit MIT Lizenz und kostenpflichtige Pro-Version	Kommerzielle Nutzung: Kostenpflichtig
Stars auf Github	1.6k	7.5k	5.6k	21.8k
Größe der Bibliothek	1.55 MB[32]	412 kB[33]	5.04 MB[34]	22.8 MB[35]
Datum des letzten stabilen Release	2025[36]	2018[37]	2026[38]	2025[39]
Unterstützte Features	Alle nötigen Features für den Tabelleneditor vorhanden	Alle nötigen Features für den Tabelleneditor vorhanden	Keine für das Coding nötige Bereichsauswahl in der kostenfreien Version	Alle benötigten Funktionen für den Tabelleneditor inklusive Formel Berechnung mit *HyperFormula*[40] vorhanden

[27] https://github.com/silevis/reactgrid

Die Bibliothek *handsontable* hat den großen Vorteil, das sie mithilfe der integrierten Bibliothek *HyperFormula* Formeln innerhalb der Tabelle unterstützt, was für die Erfüllung der Anforderung FA 2.1.6 nötig ist. Die Bibliothek hat aber den entscheidenden Nachteil, für kommerzielle Nutzung kostenpflichtig zu sein. Da QDAcity zukünftig kommerziell angeboten werden soll, würden für die Benutzung von *handsontable* zusätzliche Kosten entstehen die eine Nutzung von QDAcity verteuern würden. Um langfristige Kontinuität zu gewährleisten und aufwändige Umstellung in Zuge der geplanten Kommerzialisierung zu vermeiden wurde die Bibliothek *handsontable* ausgeschlossen.
Die Bibliothek *X Data Grid* gibt es als kostenpflichtige Pro-Version und als kostenlose und damit für QDAcity geeignete Version unter MIT Lizenz. Bei der kostenlosen Version wurde aber der Umfang der Funktionalitäten derart eingeschränkt, dass sie nicht mehr für QDAcity geeignet ist. Ein nur von der Pro-Version unterstütztes Feature, ist die Bereichsauswahl, die für das Codieren von mehreren Zellen nötig ist. Deshalb ist kostenlose Version dieser Bibliothek nicht ausreichend.[41]
React-data-grid überzeugt mit einem hohen Bekanntheitsgrad, den man an der hohen Anzahl an Stars auf Github erkennen kann, und mit einer breiten Auswahl an unterstützten Funktionalitäten, die alle frei nutzbar unter der MIT Lizenz stehen. Die letzte stabile Version der Bibliothek datiert allerdings auf das Jahr 2018 und nur Beta-Versionen dieser Bibliothek sind mit der aktuell im QDAcity-Projekt verwendeten React Version kompatibel. Beim Testen der beta-Versionen zeigten sich gravierende Fehler, wie zum Beispiel Zellen, die trotz Markierung als nicht editierbar weiter bearbeitbar waren, was die Verwendung im Coding-Editor verhindert.
Aus diesen Gründen fiel die Entscheidung zugunsten der Bibliothek *ReactGrid*, die zwar weniger bekannt ist, aber alle benötigten Features für den Tabelleneditor unterstützt, beziehungsweise den Freiraum bietet, individuell die Funktionalitäten zu erweitern. Außerdem steht sie vollständig unter MIT Lizenz und kann somit frei benutzt werden. Während der Implementierung konnte auch die gute

[28] https://github.com/silevis/reactgrid
[29] https://github.com/Comcast/react-data-grid
[30] https://github.com/mui/mui-x
[31] https://github.com/handsontable/handsontable
[32] https://www.npmjs.com/package/@silevis/reactgrid
[33] https://www.npmjs.com/package/react-data-grid
[34] https://www.npmjs.com/package/@mui/x-data-grid
[35] https://www.npmjs.com/package/handsontable
[36] https://www.npmjs.com/package/@silevis/reactgrid?activeTab=versions
[37] https://www.npmjs.com/package/react-data-grid?activeTab=versions
[38] https://www.npmjs.com/package/@mui/x-data-grid?activeTab=versions
[39] https://www.npmjs.com/package/handsontable?activeTab=versions
[40] https://github.com/handsontable/hyperformula
[41] https://v4.mui.com/components/data-grid/getting-started/#feature-comparison

Dokumentation der Bibliothek überzeugen.[42]

3.3.3 Bibliotheksvergleich für den Upload von XLSX-Dateien

XLSX-Dateien sollen in QDAcity hochgeladen und als Tabellenkalkulationsdokument behandelt werden. Dazu wird eine Bibliothek benötigt, mit der XLSX-Dateien eingelesen werden können, um diese in der im Kapitel 3.3.1 erläuterten Struktur im YDoc zu persistieren. Diese Funktionalität bieten die folgenden Bibliotheken, die in der Tabelle 3.3 gegenübergestellt werden.

Tabelle 3.3: Gegenüberstellung verschiedener Bibliotheken zum Lesen der XLSX-Dateien

Bibliothek	exceljs[43]	sheetJS[44]	xlsx-populate[45]
Lizenz	MIT	Community-Version unter Apache-2.0	MIT
Stars auf Github	15.1k	36.2k	994
Größe der Bibliothek	21.8 MB[46]	1.39kB[47]	15.1 MB[48]
Art der Installation	NPM-Registry[49]	Als Tarball[50]	NPM-Registry[51]
Alter der aktuellsten Version	2 Jahre[52]	2 Jahre[53]	6 Jahre[54]

Aufgrund der, unter anderem an den 994 Stars auf Github erkennbar, geringen Bekanntheit und dem hohen Alter von 6 Jahren der aktuellsten Version, wurde die Bibliothek *xlsx-populate* ausgeschlossen.
SheetJS kann mit seiner hohen Anzahl an Stars auf Github und der geringen Größe der Bibliothek überzeugen. Sie hat aber den großen Nachteil, dass die neueren Versionen der Bibliothek nicht über das in QDAcity verwendete *npm-Registry*

[42] https://silevis.github.io/reactgrid/docs/4.0/1-getting-started
[43] https://github.com/exceljs/exceljs
[44] https://git.sheetjs.com/SheetJS/sheetjs
[45] https://github.com/dtjohnson/xlsx-populate
[46] https://www.npmjs.com/package/exceljs
[47] https://www.npmjs.com/package/sheetjs
[48] https://www.npmjs.com/package/xlsx-populate
[49] https://github.com/exceljs/exceljs?tab=readme-ov-file#installation
[50] https://docs.sheetjs.com/docs/getting-started/installation/frameworks
[51] https://github.com/dtjohnson/xlsx-populate?tab=readme-ov-file#browser
[52] https://www.npmjs.com/package/exceljs?activeTab=versions
[53] https://git.sheetjs.com/sheetjs/sheetjs/tags
[54] https://www.npmjs.com/package/xlsx-populate?activeTab=versions

in das Projekt geladen werden können, sondern ausschließlich über ein *Tarball* installiert werden können. Dies erschwert die Wartbarkeit, weil neue Versionen dadurch nicht über den Steuerbefehl *npm update* geladen, sondern manuell in das Projekt eingefügt werden müssen. Deshalb fiel die Entscheidung auf *exceljs*, eine bekannte Bibliothek unter MIT Lizenz, die über die *npm-Registry* in das Projekt geladen werden kann.

3.3.4 Der neue Coding-Type *CellCoding* für Tabellen

Für das Codieren von Tabellen sind die vorhandenen Coding-Typen für Bereiche und Texte nicht geeignet, denn in den Tabellen sollen bestimmte Zellen codiert werden können. Deshalb wurde die neue Klasse *CellCoding* als Unterklasse der abstrakten Klasse *BaseCoding* der im Kapitel 3.1.3 erläuterten Vererbungshierarchie der Coding-Typen hinzugefügt. Die neue Klasse *CellCoding* erweitert die

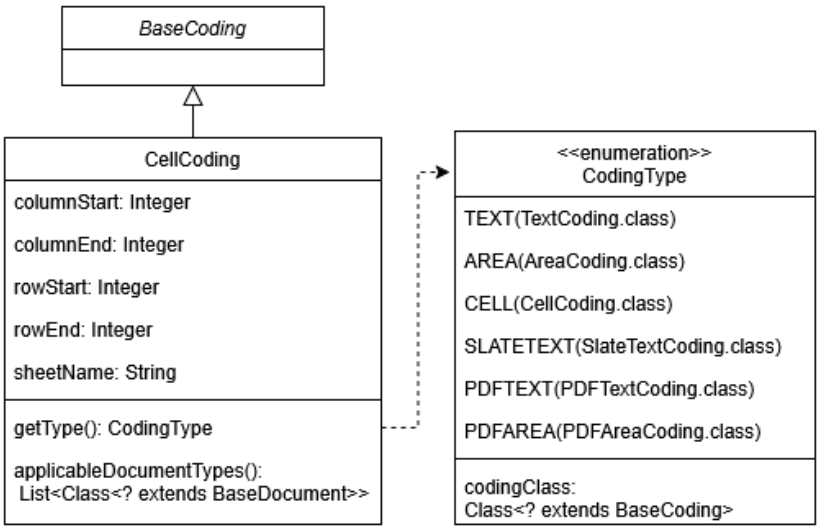


Abbildung 3.8: Klassendiagramm für die neue Klasse *CellCoding*

Klasse *BaseCoding* um die Attribute *columnStart, columnEnd, rowStart* und *rowEnd*, die den Bereich der codierten Zellen begrenzen. Außerdem wird in *sheetName* der Name des Tabellenblatts gespeichert. Die Methode *applicableDocumentTypes* liefert eine Liste mit der Dokumentarten, die mit diesem Coding-Typ codiert werden können. Der Rückgabewert der Methode ist eine Liste mit dem Eintrag *SpreadsheetDocument.class*, die im Kapitel 3.3.1 vorgestellt wurde. Die Methode *getType* liefert den *CodingType CELL* zurück. Diese Bezeichnung wurde mit der *codingClass CellCoding.class* im Enum *CodingType* hinzugefügt.

4 Design und Implementierung

4.1 Vorgenommene Erweiterungen für die Unterstützung des Dateiformats DOCX

4.1.1 Erweiterung des Texteditor für den Upload von DOCX-Dateien

In diesem Kapitel werden die grundlegenden Implementierungsschritte vorgestellt, die nötig waren, damit eine einfache DOCX-Datei hochgeladen werden kann.

Erweiterung der unterstützen Dateiformate

Um DOCX-Dateien hochladen zu können, muss der Mime-Typ dieses Dateityps *application/vnd.openxmlformats-officedocument.wordprocessingml.document* freigeschaltet werden.[1] Da das hochgeladene Dokument als Textdokument behandelt werden soll, muss es in der Liste der unterstützten Mime-Typen von Textdokumenten hinzugefügt werden. Zur Bereitstellung dieser Liste im Frontend gibt es für jede Dokumentart, die im Backend als Unterklasse der Klasse *BaseDocument* modelliert und im Klassendiagramm im Kapitel 3.1.2 visualisiert ist, die Methode *getSupportedMimeTypes*, die in den Unterklassen eine Liste der unterstützten Mime-Typen zurückgibt. Über den Endpunkt *getSupportedDocumentTypes* werden diese vom Frontend abgefragt, sodass beim Uploadprozess nur Dokumente mit einem unterstützen Mime-Typ zur Verfügung stehen.

Implementierung des Uploadprozesses von DOCX-Dokumenten

Die für den Uploadprozess im Frontend benötigte Implementierung, die die vorhandene *upload*-Methode im *TextEditorPlugin* erweitert, wird im folgenden beschrieben. Analog zu anderen hochladbaren Dokumentarten wird zunächst ein

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/MIME_types/Common_types

neues leeres Dokumentskelett für ein Textdokument angelegt und dessen Metadaten gesetzt. Dabei wird die Dokumentversion direkt auf den Wert 2 festgelegt. Diese Versionsnummer gibt an, ob das CES den Text noch als Blob in den GCS hochladen muss. Da dieser Umweg über Backend ins CES für den neuen DOCX-Datei-Import nicht genommen werden soll (vgl. NFA 1.1), wird die Versionsnummer direkt auf 2 gesetzt und das Blob ohne Umweg im Frontend persistiert. Deshalb wird auch der *Content* des Textdokument nicht gesetzt, den das CES bei anderen Textimporten konvertieren würde. Bevor der Inhalt abgespeichert wird, muss dieser in Slate-Nodes umgewandelt werden. Dazu wird die Javascript-Bibliothek Mammoth (vgl. Kapitel 3.2.1) dynamisch geladen, die die hochgeladene DOCX-Datei in HTML umgewandelt und in einen *ArrayBuffer* gespeichert. Wie in Kapitel 3.2.2 beschrieben, wird der HTML-Code aus dem ArrayBuffer zu Slate-Nodes konvertiert und als Blob im GCS gespeichert, siehe *Kapitel 3.2.3*. Anschließend wird das Dokument im Texteditor angezeigt und kann dort wie alle anderen Textdokumente bearbeitet und codiert werden.

4.1.2 Erweiterung der Konvertierung für komplexe DOCX-Dateien

Um auch komplexere DOCX-Dateien anzeigen zu können, muss die vorhandene Konvertierungslogik sowohl von HTML zu Slate-Nodes als auch beim Rendern von Slate-Nodes zu HTML ergänzt werden. Ursprünglich enthielt diese Logik nur die Funktionalitäten für Listen, unterstrichene, kursive und fettgedruckte Zeichen, was für komplexere Formatierung nicht ausreicht. In der *SlateUtils* Datei werden alle Elemente des HTML-Textes rekursiv zu Slate-Nodes deserialisiert. Wie im *Kapitel 3.2.2* beschrieben, wird mit einer *switch-case-Struktur* abhängig von dem HTML-Tag das entsprechende Slate Node eingefügt und dadurch eine Baumhierarchie aufgebaut, die die HTML-Struktur widerspiegelt. Um zusätzliche HTML-Tags zu unterstützen, müssen neue *switch-case-Zweige* eingefügt werden, die die entsprechenden HTML-Tags zu Slate-Knoten umwandeln. Damit diese dann wieder im Texteditor angezeigt werden können, müssen die Methoden *renderLeaf*, die Textknoten von Slate zu einer HTML-Struktur zusammenbaut, und *renderElement*, die die Container-Elemente zu HTML-Tags umwandelt, in der Klasse *TextEditorCollaborative* erweitert werden. Diese Methoden rendern das Textdokument im Texteditor.

Erweiterung der Zeichenformatierung

Mammoth kennzeichnet durchgestrichene Buchstaben mit den HTML-Tag *s*, welches als neuer *switch-case-Zweig* in die Deserialisierungsfunktion eingefügt wurde. Alle Kind-Elemente dieses Tags werden als Text-Knoten mit dem entsprechenden Attribut *strikethrough* gespeichert. Gleichzeitig muss dieses Attribut auch beim Rendern des Textdokuments berücksichtigt werden. Dazu wird die *render-*

Leaf Methode erweitert. Wenn das entsprechende Attribut im Textknoten gesetzt wurde, wird ein *s-Tag* in die HTML-Hierarchie eingefügt, damit der Text durchgestrichen angezeigt wird. Für die HTML-Tags *sub* und *sup* zur Darstellung von hoch- und tiefgestellten Buchstaben wurde analog vorgegangen. Standardmäßig ignoriert *Mammoth*, ob Text unterstrichen ist oder nicht.[2] Um dieses Verhalten zu ändern, kann man *Mammoth* für die Konvertierung von DOCX-Dokuments zu HTML eine individuell konfigurierbare Liste an Umformungen mitgeben, die dann von *Mammoth* berücksichtigt wird. Wenn diese Liste die Zuordnung *u => u* enthält, wird jedem unterstrichenen Text in der DOCX-Datei das HTML-Tag *u* zugewiesen, sodass diese dann analog zu den anderen Zeichenformatierungen berücksichtigt werden.

Hierarchische Überschriften

Eine DOCX-Datei kann eine Hierarchie an Überschriften enthalten, die dann auch als solche im Texteditor angezeigt werden soll. Dafür konvertiert *Mammoth* die Überschriften zu den Überschriftelementen *h1* bis *h6*, wobei Letzteres die kleinst mögliche Hierarchieebene in HTML ist (Bühler et al., 2023). In Slate wird der Inhalt einer solchen Überschrift als Kind-Element des Container-Elements vom Typ der jeweiligen Hierarchiestufe gespeichert. In der *renderElement*-Methode wird dann beim Rendern des Container-Elements das jeweils entsprechende HTML-Tag *h1* bis *h6* eingefügt.
Sowohl bei den mit Google Docs als auch bei den mit Word erzeugten DOCX-Dateien kann zusätzlich zu Überschriften auch Text als *Title* markiert sein. Damit auch diese Formatierungen im Texteditor von QDAcity übernommen wird, muss die individuell konfigurierbare Liste für die Konvertierung um die folgende Umformung ergänzt werden: *p[style-name='Title'] => h1.titel*. Dadurch werden Textabschnitte, die in DOCX als *Title* markiert sind, als *h1* HTML-Tags behandelt.

Tabellen

DOCX-Dokumente können außerdem Tabellen erhalten, die im Texteditor von QDAcity angezeigt werden sollen. Für die Konvertierung von Tabellen werden die entsprechenden HTML-Tags *table, thead, tbody, tr* und *td* verwendet. Diese werden analog zu den hierarchischen Überschriften als Slate-Container-Element-Knoten mit dem entsprechender Tag-Bezeichnung als Knoten-Typ gespeichert. Mit der *renderElement*-Methode werden die Container-Elemente bei der Darstellung der Textdatei dann wieder in die HTML-Hierarchie umgewandelt, sodass die Tabellenstruktur beibehalten wird. Im React Projekt wird die Bibliothek *styled-components*[3] verwendet, um das CSS mit den Style-Vorgaben für die

[2] https://github.com/mwilliamson/mammoth.js?tab=readme-ov-file#underline
[3] https://www.npmjs.com/package/styled-components

React-Komponenten festzulegen. Das Design der Tabellenkomponenten wird daher ebenfalls mithilfe dieser Bibliothek bestimmt.

Links

Bei der Konvertierung von Links muss zwischen Links, die im Text eingebettet sind, und Links, die in einem eigenem Absatz sind, unterschieden werden, da sie zu unterschiedlichen Slate-Nodes umgeformt werden müssen. Eingebettete Links sind daran zu erkennen, dass das DOM-Element mit dem HTML-Tag *a* mindestens ein Geschwister-Element hat. Um sicherzustellen, dass bei dem in QDAcity importierten Textdokument hinter einem eingebetteten Link im selben Abschnitt weiterer Text geschrieben werden kann, muss vor der Konvertierung zu Slate-Nodes in die HTML-Struktur nach dem Link, falls zwar ein Geschwisterknoten davor aber keines danach existiert, ein leeres Textelement eingefügt werden. Das eingebettete Link-Element und alle seine Kind-Elemente werden dann zu Slate-Textknoten umgeformt, die als Attribut den *url-Link* bekommen. Die Geschwisterknoten werden durch die rekursive Deserialisierung entsprechend ihres HTML-Tags konvertiert. In der *renderLeaf*-Methode werden diese dann bei der Ausgabe des Textes wieder zum HTML-Tag *a* mit dem *href* des entsprechenden *url-Links* gerendert.
Bei einem Link in einem eigenen Absatz der DOCX-Datei wird das HTML-Tag in ein Container-Element des Typs *link* konvertiert, der sowohl die Ziel-URL als auch den URL-Text speichert. Diese werden für die Ausgabe des Textes in der *renderElement*-Methode zu einem HTML-Tag umgeformt, bei dem die Ziel-URL als *href* und der URL-Text als Kind-Element gerendert werden. Außerdem wird der Texteditor mit einem Plugin so erweitert, dass die Link-Elemente als *void*- und *inline*-Elemente konfiguriert werden. Diese beiden Verhaltenstypen legen fest, wie der Slate-Editor die Link-Elemente korrekt verarbeitet. *Inline*-Elemente können als Geschwisterelemente weitere *inline*-Elemente und Textknoten haben und *void*-Elemente sind atomar, sodass dessen Kindelemente innerhalb der *renderElement*-Methode behandelt werden müssen.[4]
Die Dokumentation der Mammoth-Bibliothek[5] weist darauf hin, dass sie keine Bereinigung der DOCX-Dateien vornimmt, sodass Links mit Javascript-Zielen eingeschleust werden können und somit ein Risiko für XSS-Angriffe entsteht. Um dies zu verhindern werden nur Zieladressen eines Links gespeichert, die mit *http* oder *https* beginnen. Bei allen anderen wird das Ziel des Links auf # gesetzt, sodass der Link auf die geöffnete Seite verweist.

[4] https://docs.slatejs.org/api/nodes/element
[5] https://github.com/mwilliamson/mammoth.js?tab=readme-ov-file#security

Bilder

Die Bibliothek *Mammoth* bietet die Möglichkeit, bei der Umwandlung von DOCX zu HTML eine Funktion zur Bildkonvertierung mitzugeben. Diese wird für jedes Bild, das in der DOCX-Datei eingebettet ist, aufgerufen.[6] Die Funktion wurde von mir so implementiert, dass das Bild Base64-kodiert ausgelesen wird und daraus zusammen mit dem Mime-Typ eine Data-URL erzeugt wird. Auf dieser Basis wird ein Blob erzeugt, welches dann in ein File-Objekt umgewandelt wird. Um dieses File-Objekt in den GCS hochzuladen, wurde die Methode *uploadImageBlob* implementiert, die als Rückgabewert die *imageId* liefert (vgl. Kapitel 3.2.3). Diese wird als *src* zusammen mit dem Alternativtext (*alt*) als Attribut des *img* HTML-Tag gespeichert.
Die von *Mammoth* erzeugte HTML-Struktur muss dann vor der Konvertierung zu Slate-Nodes noch hinsichtlich korrekter Darstellung der eingebetteten Bilder bereinigt werden. Zunächst müssen die *img*-Elemente aus den Inline-Tags (*em, i, strong, b, u* und *span*) extrahiert werden, um ungültige Verschachtlungen zu vermeiden. Außerdem kann durch die Konvertierung ein Paragraph-Element *p* mit dem Bild als einziges Kind-Element entstehen. Das Paragraph-Element *p* muss manuell entfernt werden. Der letzte Schritt der Bereinigung fügt ein neuen Paragraph mit einem leeren Textfeld hinter das Bild ein, wenn dieses das letzte Element der HTML-Struktur ist. Das stellt sicher, dass unter dem letzten Bild im Texteditor neuer Text geschrieben werden kann.
Nun kann das Bild zu einem Slate-Container-Element des Typs *image* mit den Attributen *imageId* und *alt*, welches den Alternativtext beinhaltet, umgewandelt werden.
In einer imageMap werden für alle in einem bestimmten Dokument eingebetteten Bilder zur jeweiligen *imageId* die *Objekt-URLs* gespeichert. Um das Herunterladen und Verwalten dieser vom *TextEditorCollaboration*, der unter anderem die Slate-Nodes rendert, zu separieren, wurde dafür der Kontext *ImageContext* angelegt. Im zugehörigen *ImageProvider* wird die *imageMap* erstellt und verwaltet. Dazu wird über die Methode *downloadImageBlobs*, die im Kapitel 3.2.3 näher erläutert wurde, die Blobs der Bilder geladen. Um Bilder als HTML-Elemente rendern zu können, wird als *src* eine URL für diese benötigt. Deshalb wird im *ImageProvider* mithilfe der Methode *URL.createObjectURL*[7] für jeden Blob eine Objekt-URL erzeugt, die mit der *imageId* als *imageMap* zurückgegeben wird. Wenn die erstellten URLs der Bilder nicht mehr benötigt werden, werden diese im *ImageProvider* wieder freigegeben. Mit dem Aufruf *useImageMap* wird die *imageMap* dann im Texteditor in der Methode *renderElement* geladen. Mithilfe der enthaltenen Zuordnung kann dann das Container-Element des Typs *image* mit dem HTML-Tag *img* mit der Objekt-URL des zugehörigen Bildes als *src*-Attribut und dem Alternativtext als *alt*-Attribut gerendert werden. Damit der

[6] https://github.com/mwilliamson/mammoth.js?tab=readme-ov-file#image-converters
[7] https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL_static

Texteditor mit dem Bild richtig umgeht, umschließt das *img*-Element die Komponente *SyledNoneSelect*. Diese stellt mit dem HTML-Attribut *contentEditable*, welches auf *false* gesetzt ist, und der CSS Eigenschaft *user-select: none* sicher, dass der Benutzer das Bild im Editor nicht inhaltlich bearbeiten kann. Für diesen Zweck wurde zusätzlich der Texteditor mit einem Plugin so konfiguriert, dass Elemente des Typs *image void*-Elemente, also atomare Elemente, sind.

4.2 Verbesserungen für die Unterstützung des Dateiformats RTFs

Für die Umwandlung der RTF-Dateien zu HTML wird in QDAcity die Java Bibliothek *Apache Tika*[8] verwendet. Um diese auf den aktuellsten Stand zu bringen und dadurch die neusten Fehlerkorrekturen und Sicherheitsupdates zu erhalten, wurde die Bibliothek auf die Version 3.2.3 aktualisiert.
Außerdem wurde ein Fehler in dem Konvertierung-Prozess behoben. RTF-Dateien, die zum Beispiel mittels Google Docs[9] generiert wurden, enthalten am Ende des Textes eine Liste von Formatvorlagen für das Dokument als *Stylelist*. Diese *Stylelist* wird von der Bibliothek *Tika* nicht als Metadaten erkannt und wird deswegen als normaler Text im Dokument angezeigt. Deshalb muss vor der Umwandlung in der RTF-Datei nach dem Steuerwort *latentstyles* gesucht werden und der zugehörige Block komplett entfernt werden. Dadurch wird die RTF-Datei ohne die Liste an Formatvorlagen zu HTML konvertiert und gespeichert, sodass der Nutzer nur noch den eigentlichen Text sieht.

4.3 Implementierung eines neuen Tabelleneditors für Tabellenkalkulationsdokumente

Um die unterschiedlichen Dokument-Typen im Coding-Editor von QDAcity zu betrachten, zu bearbeiten und zu codieren, gibt es jeweils spezifische Editoren, die über ein Plugin-System verwaltet werden. Für Tabellenkalkulationsdokumente wurde der neue *SpreadsheetEditorCollaborative* implementiert.

4.3.1 Synchronisation des YDocs für Tabellenkalkulationsdokumente im *Hocuspocus-Server*

Analog zum Speichern von Textdokumenten wird der Inhalt eines Tabellenkalkulationsdokuments in einem YDoc gespeichert (vgl. Kapitel 3.3.1), das im GCS persistiert ist. Wie in Kapitel 3.1.1 erläutert, werden Änderungen an YDocs im

[8] https://tika.apache.org/
[9] https://docs.google.com

CES synchronisiert. Für den neuen Dokumenttyp wurde der *DocumentHandler SpreadsheetDocHandler* implementiert, der dem *YDocType SPREADSHEET* zugeordnet ist. Dieser spezifiziert die *Lifecyle-Hooks* für die YDocs, in denen die Tabellenkalkulationsdokumente persistiert sind. *OnAuthenticate* prüft mit der in QDAcity verwendeten rollen-basierten Zugriffskontrolle, ob der Nutzer autorisiert ist, das YDoc zu lesen oder zu bearbeiten. Außerdem enthält der *SpreadsheetDocHandler* die asynchrone Methode *onLoadDocument*, die zunächst das *SpreadsheetDocument* mit den Metadaten vom Backend in den CES lädt und anschließend den synchronisierten Stand des YDocs mit dem Inhalt des Tabellenkalkulationsdokuments mithilfe einer signierten Download-URL, die der Service vom Backend erhält, aus dem GCS in den CES herunterlädt. Das Speichern des YDocs im GCS wird über den *Hook onStoreDocument* implementiert. Durch die *Hocuspocus-Verbindung* kann im Frontend über den*Hook useHocuspocusProvider* auf das synchronisierte YDoc zugegriffen werden.

4.3.2 Tabellenvisualisierung mit *ReactGrid*

Für Tabellenkalkulationsdokumente wurde die Komponente *SpreadsheetEditorCollaborative* implementiert, die als Plugin-Komponente des Coding-Editors registriert wurde und für die Darstellung, Bearbeitung und das Codieren des Dokuments verantwortlich ist. Für die Visualisierung der Tabelle wird, wie im Kapitel 3.3.2 erläutert, die Bibliothek *ReactGrid* genutzt. Diese erwartet die zu visualisierende Tabelle aufgeteilt in die Attribute *rows* und *columns*. *Columns* muss ein Array von Spalten sein, die jeweils aus der Id und der Breite der Spalte bestehen. Außerdem legt der boolean-Wert *resizable* fest, ob die Breite veränderbar ist.[10] Auch die Prop *rows* muss ein Array sein, bei dem jeder Eintrag eine Map ist, die die Id der Zeile, deren Höhe und das Array mit den Zellen enthält.[11] Für jede Zelle muss, neben dem als String hinterlegten Inhalt, der Typ der Zelle angeben sein, der definiert, wie sie gerendert wird.[12]
In der Komponente *SpreadsheetEditorCollaborative* wird das YDoc des Tabellenkalkulationsdokuments mithilfe des *Hooks useHocuspocusProvider* geladen. Damit die Komponente seiteneffektfrei bleibt, wird ein *useEffect-Hook* definiert, der vom YDoc abhängig ist. Dieser wird deshalb erst aufgerufen, wenn das Rendern der Komponente abgeschlossen ist und sich das als abhängig definierte YDoc ändert (Hartmann & Zeigermann, 2020). Innerhalb des *Hooks* wird ein *observeDeep-Listener* auf das YDoc registriert, der Änderungen im Zustand erkennt und solange aktive bleibt, bis er beim *Unmouten* der Komponente wieder getrennt wird. Der *Observer* ruft die Funktion *yDocToReactGrid* auf, die den aktuellen Zustand des YDocs in die für die *ReactGrid* notwendige Datenstrukturen mit *rows* und *columns* umwandelt. Innerhalb des *Observers* werden diese beiden Attribute mit-

[10] https://silevis.github.io/reactgrid/docs/4.0/7-api/0-interfaces/3-column
[11] https://silevis.github.io/reactgrid/docs/4.0/7-api/0-interfaces/2-row
[12] https://silevis.github.io/reactgrid/docs/4.0/7-api/0-interfaces/4-cell

hilfe zweier State-Hooks, die den lokalen State innerhalb von Komponenten verwalten, gespeichert (Springer, 2020). Da durch die *Hocuspocus-Verbindung* das YDoc in Echtzeit synchronisiert wird und durch ein *ObserveDeep-Listener* jede Zustandsänderung eine Konvertierung auslöst, enthalten die beiden *States rows* und *columns* immer den aktuellen Stand der Tabelle, auch wenn mehrere Nutzer gleichzeitig an dem Tabellenkalkulationsdokument arbeiten.
Bei der Umwandlung innerhalb der Methode *yDocToReactGrid* werden zusätzlich zu den im YDoc gespeicherten Zellen auch eine Zeilen- und Spaltenüberschrift in den *rows* und *columns* mit dem Zell-Typ *Header*[13] gespeichert. Neben einer visuellen Abgrenzung zu normalen Zellen durch das Setzen der Hintergrundfarbe auf einen Grauwert aus der QDAcity-Farbpalette zeichnet sich dieser Zell-Typ auch dadurch aus, dass die Zellen nicht editierbar sind. Außerdem wird im CSS der z-Index dieses Zellentyps auf 1 festgelegt, um ein visuelles Selektieren durch eine Blaufärbung dieser Zellen zu verhindern. Analog zu den meistgenutzten Tabellenkalkulationsprogrammen werden die Zeilen von 1 bis n durchnummeriert und die Spalten erhalten eine alphabetische Nummerierung beginnend mit A. Der Zell-Typ der anderen Zellen ist abhängig vom Modus des Coding-Editors. Im Editier-Modus soll die Tabelle bearbeitbar sein, wofür der Zell-Typ *muliline-text* implementiert wurde, der im Kapitel 4.3.4 näher beleuchtet wird. Im Codier-Modus soll die Tabelle schreibgeschützt sein, was durch den selbst implementierten Zell-Typ *readonly-text* sichergestellt wird. Der Wert der Zelle ist, wie im Kapitel 3.3.1 erläutert, im YDoc in der zur Zelle gehörenden *YMap* gespeichert. Aus dieser *YMap* wird auch die Breite der Zelle entnommen und, falls sie dort noch nicht definiert ist, auf den Standartwert 120px gesetzt. Alle Zellen einer Spalte haben immer die selbe Breite, die bei der Umwandlung in dem *State columns* für jede Spalte persistiert wird. Durch das Setzen des Boolean-Werts *resizable* in der Spaltendefinition auf *true* wird festgelegt, dass die Spaltenbreite variiert werden kann. Das Ergebnis des gerenderten *ReactGrids* mit den aus dem YDoc konvertierten *rows* und *columns* ist in der Abbildung 4.1 dargestellt.

	A	B	C
1	A	B	C
2	D	E	F
3	G	H	I

Abbildung 4.1: Visualisierung der Tabelle

[13] https://silevis.github.io/reactgrid/docs/4.0/4-cell-templates/6-HeaderCell

Damit der Nutzer die Breite der Spalten ändern kann und diese Änderung auch persistiert wird, wird der *ReactGrid* die *Callback*-Funktion *handleColumnResize* als Prop übergeben, die aufgerufen wird, wenn der Nutzer durch Verschieben der Spaltenbegrenzung die Breite der Spalte ändert. Durch die übergebenen Parameter Breite und Id der Spalte kann diese Funktion den geänderten Breitenwert in den jeweiligen *YMaps*, die den Zellen der betreffenden Spalte im YDoc zugeordnet sind, speichern. Da diese Änderung des YDocs den Observer auslöst, der die Funktion *yDocToReactGrid* aufruft, werden die *States columns* und *rows* aktualisiert, und die Tabelle wird mit der neu definierten Breite der Spalte gerendert.

4.3.3 Bearbeiten einer Tabelle

Neben dem Codieren ist das Bearbeiten der Dokumente einer der wichtigsten Funktionalitäten im Coding-Editor. Deshalb muss im Editier-Modus das Tabellenkalkulationsdokument bearbeitbar sein. Dazu kann der *ReactGrid*-Komponente eine *Callback*-Funktion als Prop übergeben werden, die aufgerufen wird, sobald der Inhalt einer Zelle geändert wird. Als *Callback*-Funktion wurde die Methode *handleChanges* implementiert, die als Parameter die Änderung des Zelleninhalts und die betroffene Zelle erhält. Innerhalb einer Transaktion[14] wird die Zelle im YDoc geändert, indem mit der Id der Zeile und der Spalte auf die YMap der Zelle zugegriffen und deren Wert durch den geänderten Wert ersetzt wird. Diese Änderung am Zustand des YDocs löst durch den Observer die Konvertierung des YDocs in *rows* und *columns* aus. Dadurch sieht nicht nur der Bearbeiter der Tabelle die Änderung direkt, sondern auch Nutzer, die gleichzeitig am selben Dokument arbeiten.
Außerdem wird der *ReactGrid*-Komponente über die Prop *onContextMenu* ein Kontext-Menü übergeben. Dieses erweitert die bereits von der Bibliothek zur Verfügung gestellten Standard-Operationen Kopieren, Ausschneiden und Einfügen unter anderem um die Funktionalitäten Zeile hinzufügen, Spalte hinzufügen, Zeile löschen und Spalte löschen.[15] In der Abbildung 4.2 sieht man das geöffnete Kontext-Menü mit den verschieden Optionen im Editier-Modus des Coding-Editors. Das Kontext-Menü wird über Rechtsklick auf eine Zelle geöffnet, die dann auch als Bezugspunkt für die Operationen dient. Eine neue Zeile wird unter dieser Zelle hinzugefügt, eine neue Spalte rechts davon. Beim Löschen dieser Zeile wird die gesamte Zeile der ausgewählten Zelle gelöscht. Genauso wird beim Löschen der Spalte die gesamte Spalte, die die Zelle beinhaltet, gelöscht. Diese Operationen wurden auch als Transaktion im YDoc implementiert, sodass die Änderung der Tabelle direkt über die *Hocuspocus*-Verbindung mit allen Nutzern synchronisiert wird und der *Observer* auf dem YDoc aktiviert wird. Um diese Implementierung von der *SpreadsheetEditorCollaborative*-Komponente zu trennen,

[14] https://beta.yjs.dev/docs/api/transactions/

[15] https://silevis.github.io/reactgrid/docs/4.0/2-implementing-core-features/5-context-menu

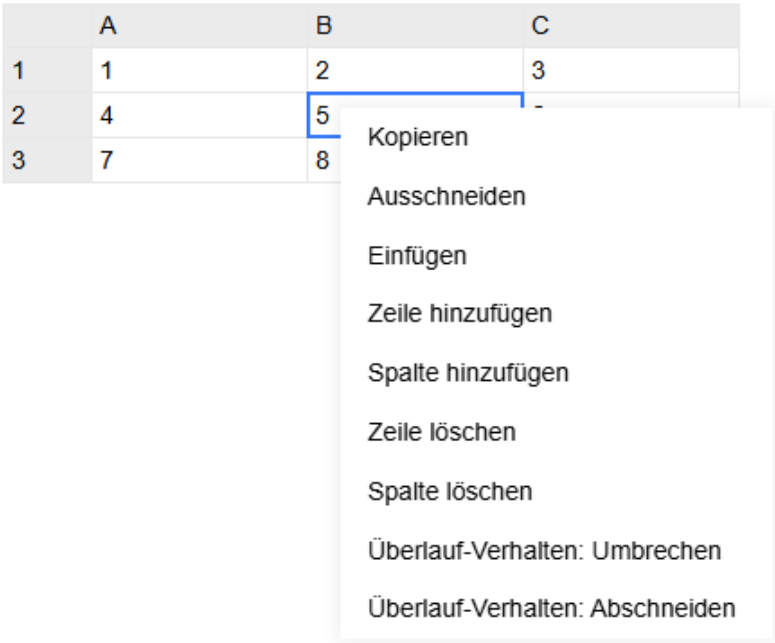


Abbildung 4.2: Geöffnetes Kontext-Menü

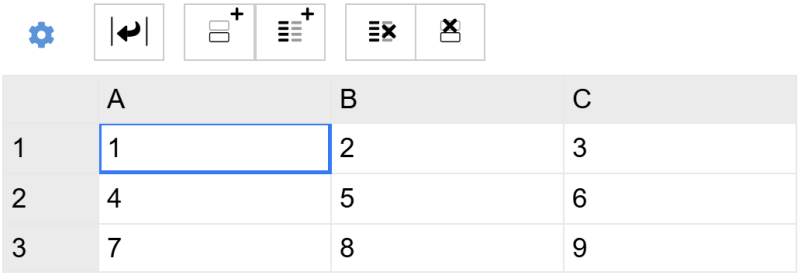


Abbildung 4.3: Toolbar zum Bearbeiten der Tabelle

wurde die Logik in einem *Context-Provider* spezifiziert, auf dessen Kontext die Komponente über den *Context-Hook useSpreadsheet* zugreifen kann (Springer, 2020).
Neben der Bearbeitung der Tabelle über ein Kontext-Menü kann der Nutzer auch die implementierte Toolbar verwenden, die in Abbildung 4.3 zu sehen ist. Für die Toolbar wurden eigene Icons designt, indem jeweils mehrere Icons der *Fontawesome*-Bibliothek[16] übereinandergelegt wurden. Über diese Icons hat man ebenfalls die Möglichkeit, die gerade ausgewählte Zeile oder Spalte zu löschen, wofür in der *SpreadsheetEditorToolbar*-Komponente auch mit dem *Context-Hook useSpreadsheets* auf die Methoden zum Löschen zugegriffen wird. Beim Klick auf das Icon zum Hinzufügen von Zeilen öffnet sich ein Drop-Down Menü mit den Op-

[16]https://fontawesome.com/v5/icons

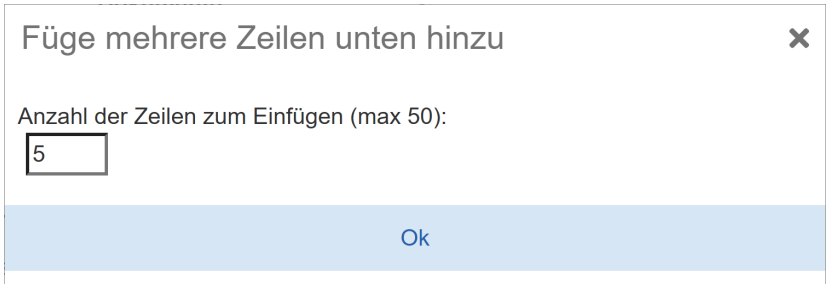


Abbildung 4.4: Modaldialog zum Festlegen der Anzahl der Zeilen zum Hinzufügen

tionen, eine Zeile über der ausgewählten Zelle hinzuzufügen oder unterhalb und für beide Varianten auch die Möglichkeit mehrere Zeilen hinzuzufügen. Wenn mehrere Zeilen ausgewählt wurden, öffnet sich ein Modaldialog mit einem Input-Feld, in dem der Nutzer die genaue Anzahl der Zeilen festlegen kann und dass in Abbildung 4.4 zu sehen ist. Diese Anzahl ist auf maximal 50 beschränkt, um die Performanz beim Rendern nicht unnötig zu belasten. Intern wird dafür auf die Methode *handleAddRow* des Kontextes zurückgegriffen, der sowohl die Einfügerichtung als auch die konkrete Anzahl der hinzuzufügenden Zeilen übergeben wird. Analog wurde dies auch für Spalten implementiert.

4.3.4 Textumbruch Verhalten innerhalb von Zellen

Das standardmäßige Umbruchverhalten des Textes in einer Zelle orientiert sich an bekannten Tabellenkalkulationsprogrammen und schneidet den Text am Ende der Zelle ab. Dieses Verhalten kann aber vom Nutzer derart geändert werden, dass der Text innerhalb einer Textzelle umgebrochen wird. Dafür hat der Nutzer zwei Möglichkeiten. Zum einen kann der Nutzer das Kontext-Menü aus Abbildung 4.2 der betreffenden Zelle öffnen und dort das gewünschte Verhalten auswählen. Zum anderen kann er die Toolbar aus Abbildung 4.3 nutzen. Diese zeigt das aktuelle Verhalten der ausgewählten Zelle als Icon an. Beim Klick auf dieses öffnet sich ein Drop-Down-Menü, in dem das Umbruch-Verhalten für den ausgewählten Bereich verändert werden kann. Das gewünschte Verhalten wird in der YMap der entsprechenden Zellen gespeichert.
Die Bibliothek *ReactGrid* verwendet für Textzellen[17] zur Eingabe von Text ein *input*-Feld. Das hat den Nachteil, das nur einzeilige Eingaben erlaubt sind und vom Nutzer eingefügte Zeilenumbrüche ignoriert werden. Um eine Zellformatierung zu ermöglichen, die auch manuelle Zeilenumbrüche für lange Textfelder unterstützt, wurde ein neuer Zell-Typ *multiline-text* implementiert. Dieser verwendet für die Eingabe mehrzeilige *textareas* (Bühler et al., 2023). Bei Eingabe der Enter-Taste in Kombination mit Steuerung, Alt oder Shift wird ein manueller Zeilenumbruch

[17]https://silevis.github.io/reactgrid/docs/4.0/4-cell-templates/8-TextCell

im Text eingefügt. Wenn der Textumbruch für eine Zelle aktiv ist, bekommt die Zelle in der Funktion *yDocToReactGrid* den *classname text-wrapping-cell*. Da für diese Klasse im CSS die Eigenschaften *white-space: pre-wrap*[18] und *overflow-wrap: break-word*[19] definiert wurde, wird der Text innerhalb einer Zelle sowohl an den manuell gesetzten Zeilenumbrüchen als auch bei zu langen Zeilen umgebrochen. Damit alle Zeilen für den Nutzer sichtbar werden, muss die Zeilenhöhe in der Tabelle angepasst werden. Da es bei der Bibliothek *ReactGrid* nur die Möglichkeit gibt, die Höhe der Zeile als Prop mitzugeben, muss diese manuell ausgerechnet werden und kann sich nicht dynamisch am Inhalt der Zelle orientieren. Für diese Berechnung wird in der Methode *yDocToReactGrid* die Funktion *getRowHeight* aufgerufen. Diese erhält sowohl den Text als auch die Breite der Zelle und berechnet daraus die benötigte Höhe. Da Zeichen einer proportionalen Schriftart unterschiedlich breit sind, wird ein *Canvas*[20] erzeugt und in deren Kontext die Schriftart und Größe gesetzt. Dadurch kann man mit der Methode *measureText*[21], aufgerufen auf dem Kontext des *Canvas*, die Breite des übergebenen Textes berechnen, wodurch sich die Anzahl der aufgrund von Überschreitung der Zellenbreite benötigten Umbrüche bestimmen lässt. Auch die manuellen Zeilenumbrüche erhöhen die Anzahl der notwendigen Zeilen. Die Höhe der Zelle lässt sich mithilfe der folgenden Formel aus der Anzahl der benötigten Zeilen berechnen:

$$heightOfCell = Math.max(defaultCellHeight, \\ Math.ceil(totalLines * fontSizeCell * lineHeightCell + padding))$$

Diese berechnet die notwendige Höhe aus der Anzahl der benötigten Zeilen multipliziert mit Schriftgröße und Zeilenhöhe, addiert die Summe des Padding nach oben und unten und bildet das Maximum aus diesem Wert und der Standardhöhe der Zellen. Die Höhe einer Zeile ergibt sich aus der maximalen Höhe aller Zellen in dieser Zeile.

4.3.5 Umgang mit mehreren Tabellenblättern

Für das Verwalten mehrerer Tabellenblätter innerhalb eines Tabellenkalkulationsdokuments sind die lokalen States *sheetNames* und *selectedSheet* der *Spreadsheet-EditorCollaborative*-Komponente zuständig. Während im *selectedSheet* der Name des Tabellenblatts angezeigt wird, dessen Tabelle aktuell im Coding-Editor angezeigt wird, sind in *sheetNames* in Form eines Arrays alle Namen der Tabellenblätter des aktuell geöffneten Dokuments gespeichert. Dieses Array wird vom *Observer*, der auf dem YDoc registriert ist, gefüllt. Wie im Kapitel 3.3.1 erläutert, sind die einzelnen Tabellenblätter unter dem Schlüssel des Blattnamens im

[18] https://developer.mozilla.org/de/docs/Web/CSS/Reference/Properties/white-space
[19] https://developer.mozilla.org/de/docs/Web/CSS/Reference/Properties/overflow-wrap
[20] https://www.w3schools.com/jsref/dom_obj_canvas.asp
[21] https://www.w3schools.com/jsref/canvas_measuretext.asp

YDoc gespeichert. Daher wird im *Observer* der *State sheetNames* auf die Schlüssel dieser YMap gesetzt. *SelectedSheet* wird beim Öffnen des Dokuments auf das erste Tabellenblatt gesetzt. In der Funktion *yDocToReactGrid* wird sichergestellt, dass die *rows* und *columns*, die der *ReactGrid*-Komponente als Props übergeben werden, immer die Zeilen und Spalten des Tabellenblatts mit dem Namen des *selectedSheets* sind. Um das aktuell ausgewählte Tabellenblatt für den Nutzer sichtbar zu machen und einen Wechsel zwischen den Blättern zu ermöglichen, wurde eine Register-Navigation implementiert, die in der Abbildung 4.5 gezeigt ist. In dieser Navigationsleiste wird das geöffnete Tabellenblatt durch blaues un-

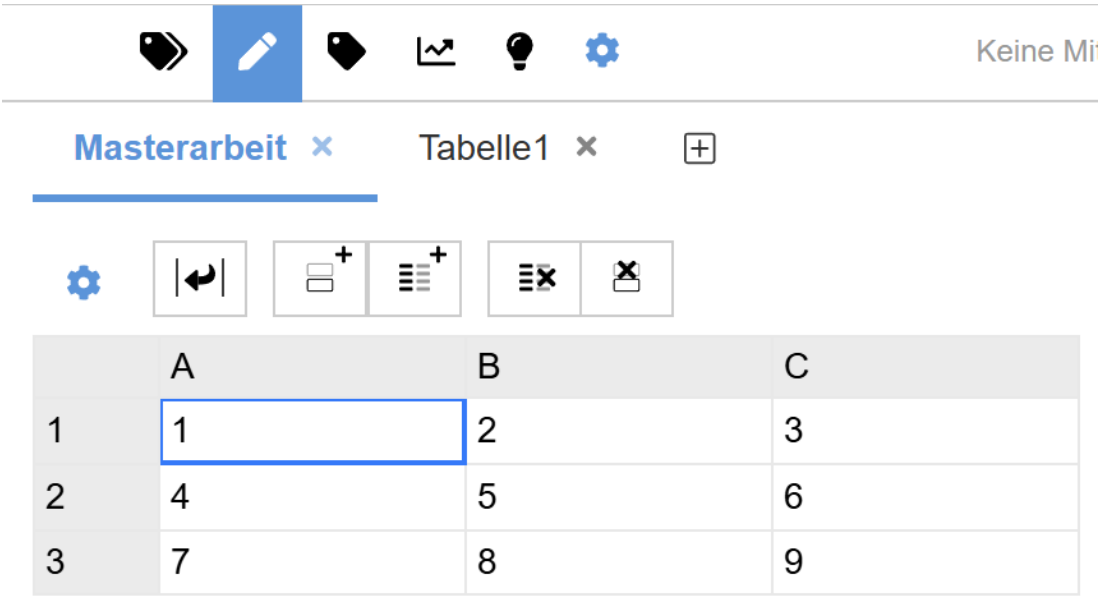


Abbildung 4.5: Navigation zwischen den Tabellenblättern

terstreichen und den blau eingefärbten Namen kenntlich gemacht. Ein Wechsel zu einem anderen Tabellenblatt ist durch das Klicken auf den entsprechenden Namen möglich. Ein Wechsel führt intern dazu, dass der Zustand von *selectedSheet* auf den neuen Tabellenblatt-Namen gesetzt wird. Im Editier-Modus, der in der Abbildung 4.5 aktiv ist, ist es außerdem möglich, Tabellenblätter zu löschen und neue anzulegen. Durch einen Klick auf das X-Symbol neben dem Namen des jeweiligen Tabellenblatts, welches aus der *FontAwesome* Bibliothek[22] stammt, kann dieses gelöscht werden. Besteht ein Tabellenkalkulationsdokument jedoch nur aus einem einzigen Blatt, dann wird kein Lösch-Icon angezeigt um sicherzugehen, dass ein Tabellenkalkulationsdokument immer mindestens ein Tabellenblatt enthält. Bevor das Tabellenblatt endgültig aus dem YDoc entfernt wird, muss das Löschen noch in einem sich öffnenden Dialog bestätigt werden. Rechts neben dem Register der Tabellenblätter befindet sich ein Button mit einem Plus-Icon,

[22]https://fontawesome.com/v5/icons

mit dem der Nutzer ein neues Blatt anlegen kann. Der Klick auf diesen Button löst die Methode *handleAddSheet* aus, die ein neues Tabellenblatt mit 50 Zeilen und 10 Spalten mit leeren Zellen im YDoc speichert. Der für das Tabellenblatt verwendete Name wird bei einer auf deutsch eingestellten Website von QDAcity aus der Bezeichnung *Tabelle* ergänzt um einen Index erzeugt. Der Index beginnt bei 1 und es wird jeweils der kleinste, noch nicht innerhalb diesen Dokuments verwendete Wert genommen. Da diese Operationen direkt innerhalb des YDocs passieren, wird der Observer bei allen Nutzern, die das Dokument aktuell geöffnet haben, ausgelöst, wodurch alle auf den selben Stand gebracht werden.

4.3.6 Erstellen eines neuen Tabellenkalkulationsdokuments

Es gibt in QDAcity zwei verschiedene Möglichkeiten ein neues Tabellenkalkulationsdokument anzulegen. Zum einen kann man ein leeres Dokument erstellen und anschließen direkt im Editor befüllen. Zum anderen kann eine XLSX-Datei hochgeladen werden. Beide Wege werden in diesem Kapitel vorgestellt.

Erstellen eines leeren Tabellenkalkulationsdokuments

Für diese Option wurde das Dropdown Menü, das im Upload-Modal in der Abbildung 4.6 zu sehen ist, um den Eintrag *Tabelle* erweitert. Dadurch kann der Nutzer über den für neue Textdokumente gewohnten Weg auch ein neues Tabellenkalkulationsdokument anlegen und dabei direkt den Namen des Dokuments festlegen. Damit dies funktioniert, wurde der *DocumentType Spreadsheet*, welches im Verantwortungsbereich der *SpreadsheetEditorPlugin*-Komponente liegt, den *SUPPORTED_FILETYPES* hinzugefügt. In dieser Komponente wird beim Erstellen eines neuen Tabellenkalkulationsdokuments ein neues *SpreadsheetDocument* mit den entsprechend Metadaten angelegt. Dieses wird über den Endpunkt *insertDocument* an das Backend geschickt und dort im *GCS* persistiert. Der *CES* stellt beim Laden des Dokuments fest, dass noch kein zugehöriges YDoc für das *SpreadsheetDocument* existiert, da er beim Versuch des Ladens aus dem GCS statt des Dokuments eine Rückmeldung mit dem HTTP-Status 404 bekommt, und initialisiert dementsprechend ein neues YDoc. In diesem wird unter dem Namen des Dokuments ein Tabellenblatt angelegt, dass eine leere Tabelle mit 50 Zeilen und 10 Spalten enthält. Dieses YDoc wird dann in dem GCS persistiert. Der Nutzer sieht die neu erstellte Tabelle und kann diese nun bearbeiten und codieren.

Upload einer XLSX-Datei

Eine andere Möglichkeit ist, das Hochladen einer XLSX-Datei über das Upload-Modal in QDAcity. Dafür wurde im *SpreadsheetDocument* der Mime-Typ fürs XLSX-Dateien als *SupportedMimeTypes* festgelegt (vgl. Kapitel 3.3.1). Für den

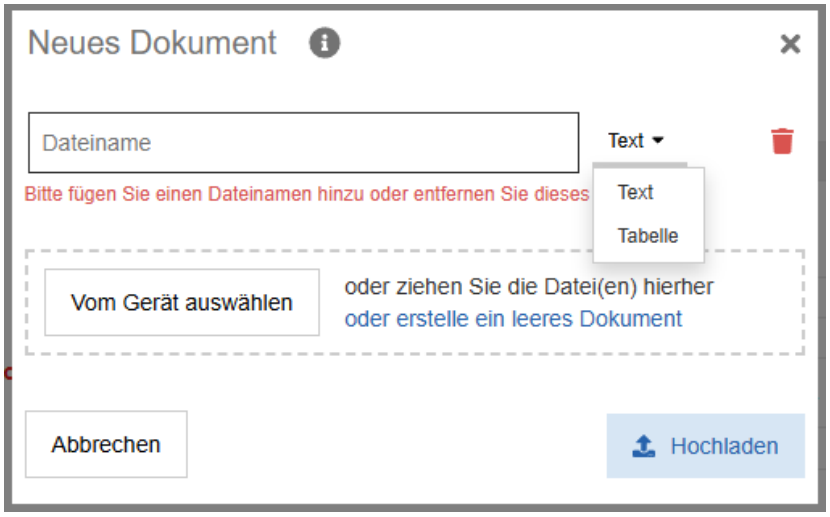


Abbildung 4.6: Upload-Modal zum Erstellen eines neuen Dokuments

Upload-Prozess ist ebenfalls die *SpreadsheetEditorPlugin*-Komponente zuständig. Diese initialisiert ein neues *SpreadsheetDocument* mit Metadaten und schickt dieses an das Backend, das es im GCS persistiert. Die hochgeladene Datei wird dann mithilfe der Bibliothek *exceljs* ausgelesen, die sich im Vergleich mit alternativen Produkten durchsetzen konnte (vgl. Kapitel 3.3.3). Die Inhalte werden dann in die im Kapitel 3.3.1 vorgestellte Speicherstruktur konvertiert. Dabei wird sichergestellt, dass aus jedem Tabellenblatt eine rechteckige Tabelle entsteht, indem kürzere Zeilen und Spalten mit leeren Zellen aufgefüllt werden. Das dadurch entstandene YDoc wird über die vorhandene Methode *uploadArray* direkt vom Frontend in den GCS geladen. Dadurch kann der CES das YDoc aus dem GCS laden und über die *Hocuspocus*-Verbindung synchronisieren. Der Nutzer sieht seine hochgeladene Datei im Coding-Editor und kann diese bearbeiten und codieren.

4.3.7 Codieren von Tabellen

Das Codieren von Dokumenten ist eine der grundlegenden Funktionen von QDAcity. Damit auch Tabellenkalkulationsdokumente codiert werden können, wurde der neue Coding-Typ *CellCoding*, der im Kapitel 3.3.4 vorgestellt wird, eingeführt. Dieser ermöglicht das Codieren eines Bereichs von Zellen, der durch die Attribute *columnStart, columnEnd, rowStart* und *rowEnd* festgelegt ist.

Neues Coding hinzufügen

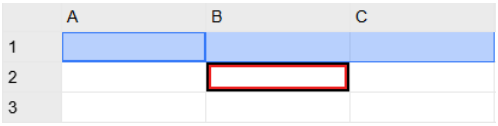
In der linken Spalte des Coding-Editors steht dem Nutzer der Button *Code anwenden* zur Verfügung, um einen im Dokument ausgewählten Bereich zu codieren. Dafür wird der *ReactGrid*-Komponente die Callback-Methode *handleSelectionChanged* übergeben, die den vom Nutzer ausgewählten Bereich der Tabelle im *State*

selectedRanges persistiert. Das Auswählen eines Bereiches wird durch die Prop *enableRangeSelection* der *ReactGrid*-Komponente freigeschaltet. Für die Anwendung eines Codes wird ein neues *CellCoding* mit den Werten für die Begrenzung des markierten Tabellenbereichs aus dem *State selectedRanges* erstellt. Dabei wird überprüft, ob *CellCodings* vom gleichen Code existieren, die im gleichen Tabellenblatt gesetzt wurden und vom neuen Coding komplett eingeschlossen werden. Solche umschlossenen Codings werden dann gelöscht. Das Anlegen, Ändern und Löschen eines *CellCoding* funktioniert analog zu den anderen *Coding-Types*. Im *CodingProvider* wird eine *Hocuspocus*-Verbindung zum CES aufgebaut und das YDoc mit den in der YMap gespeicherten Codings zurückgegeben. Innerhalb einer Transaktion werden alle Änderung der Codings im YDoc übernommen und durch die *Hocuspocus*-Verbindung synchronisiert.

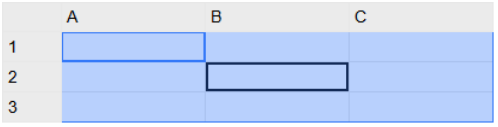
Bereich von Coding entfernen

Für die *CellCodings* musste außerdem ein Algorithmus implementiert werden, der dem Nutzer das Entfernen einzelner ausgewählter Zellen aus den Codings eines bestimmten Codes ermöglicht. Nach der Definition von *CellCodings*, die dem Klassendiagramm in Abbildung 3.8 zu entnehmen ist, müssen diese rechteckig sein. Um dies sicherzustellen, muss in manchen Fällen das Coding beim Löschen von Unterbereichen in mehrere neue Codings aufgeteilt werden. Das Entfernen eines Bereiches aus einem Coding lässt sich auf die 6 Fälle aufteilen, die in Abbildung 4.7 dargestellt sind. In den Ansichten ist der Bereich des Codings vor dem Entfernen mit einem schwarzen Rahmen und danach mit rotem Rahmen gekennzeichnet. Der zu entfernende Bereich ist blau hinterlegt.

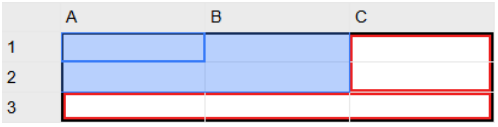
Im Fall 1 gibt es zwischen dem zu entfernenden Bereich und dem Coding keine Überlappung, sodass das Entfernen des Bereichs keine Auswirkung auf das Coding hat. Wenn der zu entfernende Bereich das Coding, wie im Fall 2, komplett umschließt, wird das Coding mit entfernt. Fall 3 tritt ein, wenn eine Ecke aus einem Coding entfernt werden soll. Dann muss das ursprüngliche Coding auf zwei neue, rechteckige Codings aufgeteilt werden, da ein Teil des ursprünglichen Coding-Bereichs ohne die entfernte Ecke schmaler ist als das restliche Coding. Im Fall 4 wird ein Bereich mittig an einer Kante des Codings entfernt. Das führt zur 3-Teilung des Codings, weil der mittige Bereich schmaler ist, als das obere und untere Coding. Ein Sonderfall von Fall 4 ist Fall 5, bei dem über die gesamte Breite des Codings ein Bereich entfernt werden soll. Wie in der Abbildung Fall 5 zu sehen, wird dafür das Coding in 2 neue Codings aufgeteilt, die sich nicht mehr berühren. Analog funktioniert eine solche Aufteilung auch bei einer vertikalen Entfernung eines sich über die gesamte Höhe des Codings erstreckenden Bereichs. Fall 6 tritt ein, wenn ein Bereich in der Mitte eines Codings entfernt werden soll, der nicht bis zum Rand des Codings reicht. Dann wird dieses Coding auf 4 Codings aufgesplittet, wobei eines über dem entfernten Bereich, eines links, eines rechts und eines unterhalb liegt. Für jedes zum Code gehörende Co-



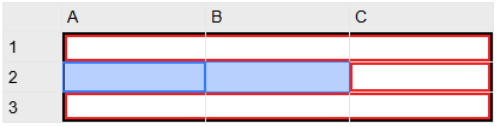
(a) Fall 1: Entfernen eines Bereichs ohne Überschneidung mit Coding



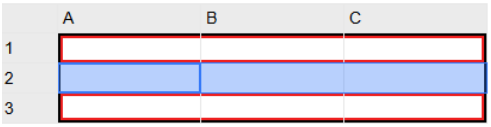
(b) Fall 2: Entfernen eines kompletten Codings



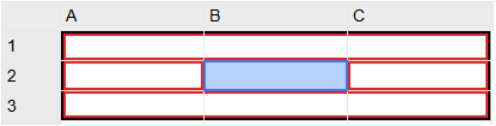
(c) Fall 3: Entfernen einer Ecke eines Codings



(d) Fall 4: Entfernen eines Bereichs mittig an einer Kante eines Codings



(e) Fall 5: Aufteilung auf zwei getrennte Codings



(f) Fall 6: Entfernen der Mitte eines Codings

Abbildung 4.7: Die 6 verschiedenen Fälle zum Entfernen eines Bereiches aus einem Coding

ding innerhalb des Tabellenblatts, in dem ein zu entfernender Bereich markiert wurde, muss überprüft werden, welcher der 6 Fälle vorliegt und dementsprechend vorgegangen werden.

Visuelle Darstellung von Codings

Für das Anzeigen von gesetzten Codings innerhalb eines Tabellenkalkulationsdokuments wurden zwei Ansichten implementiert, die entweder einzeln oder auch parallel aktiviert werden können, wie es in Abbildung 4.8 zu sehen ist.

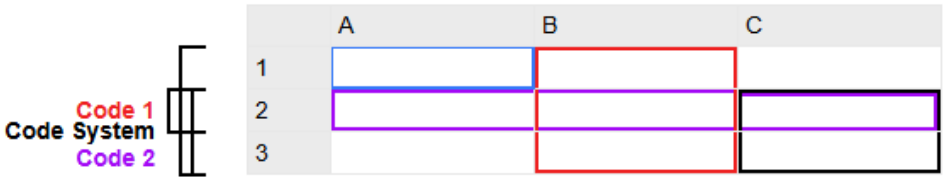


Abbildung 4.8: Visuelle Darstellung der Codings in Tabellenkalkulationsdokumente

Aktiviert wird die Visualisierung des Tabellencodings mithilfe eines DropDown Menüs mit Checkboxen, die erscheinen, sobald der Nutzer das Zahnradsymbol in der Toolbar aus Abbildung 4.3 drückt, siehe Abbildung 4.9. Die Auswahl wird in den zwei *States showCodingColumn* und *showCodingTable* verwaltet. Die

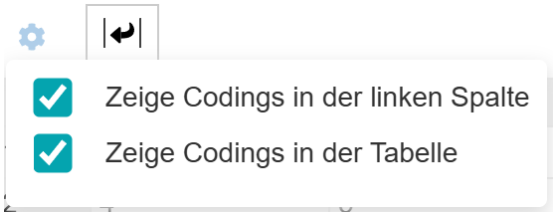


Abbildung 4.9: Einstellung der Anzeige Optionen für Codings

Codings können durch *Coding-Brackets* am linken Rand der Tabelle visualisiert werden, wie es der Nutzer bereits von den anderen Dokumentarten gewöhnt ist. Für die Anzeige bei Tabellenkalkulationsdokumenten, wird die *CodingBrackets*-Komponente wiederverwendet. Diese benötigt die Prop *CodingBracketData*, die für jedes Coding neben den Metadaten auch einen Wert für *offsetTop* mit dem Abstand zum oberen Rand des Dokuments und für *height* mit der Höhe eines Codings erfordert. Dazu werden über die Ids der Zeilen und der Spalten der Zellen in der linken oberen Ecke und der linken unteren Ecke des Codings Referenzen auf die DOM-Elemente ermittelt. Für den Wert für *offsetTop* wird die y-Koordinate des Root-Elements des Tabelleneditors von der y-Koordinate der Referenz auf die obere linke Zelle abgezogen. Die Höhe des Codings lässt sich durch die Position der unteren Kante der Zelle, in der linken unteren Ecke innerhalb des codierten Bereichs abzüglich der Position der oberen Kante der Zelle in der linken oberen Ecke berechnen. Dementsprechend werden die folgenden Formeln zur Berechnung des Offsets und der Höhe verwendet:

$$offsetTop = cellRectTopLeft.y - rootRect.y$$

$$height = cellRectBottomLeft.bottom - cellRectTopLeft.top$$

Mit diesen Werten für *offsetTop* und *height* rendert die *CodingBrackets*-Komponente die *Coding-Brackets* mit der richtigen Größe und auf Höhe der codierten Zellen. Zusätzlich wird der Komponente über die Prop *onBracketClick* die *Callback*-Funktion *setManuallyActivatedCoding* übergeben, die beim Anklicken eines *Coding-Brackets* mit der Id des entsprechenden Codings ausgeführt wird. Sie führt die Methode *setActivatedCodingId* des *Coding-Editor*-Kontext aus, welche die Methode *activateCodingInEditor* triggert. Diese sorgt dafür, dass der codierte Bereich im *State selectedRanges* persistiert wird und färbt entsprechenden die Zellen des Codings blau ein, indem sie die Zellen über die Prop *highlights* an die *ReactGrid*-Komponente übergibt. Diese Visualisierung ist jedoch vor allem bei

breiten Tabellen problematisch, da sie die für die eigentliche Tabelle zur Verfügung stehende Breite reduziert und unübersichtlich wird, wenn mehrere Codings innerhalb derselben Zeile einer Tabelle liegen und somit auf der gleichen Höhe gesetzt sind.
Deshalb wurde eine zweite Visualisierungsmöglichkeit implementiert, die Codings direkt innerhalb der Tabelle anzeigt. Bei dieser Darstellung werden die Umrisse der Codings durch *inset Box-Shadows*[23], also mit nach innen eingerückten, farbigen Schatten der Zellränder, innerhalb der Tabelle visualisiert, wie es auch in der Abbildung 4.8 zu sehen ist. Dazu wird für jede Zelle geprüft, welche Coding-Ränder mit den Rändern der Zelle übereinstimmen. Da diese Überschneidung mit mehreren Codings passieren kann, wird jeweils ein Offset berechnet, sodass die *Box-Shadows* zueinander versetzt angezeigt werden. Der Offset wird pro Kante der Zelle mit der folgenden Formel berechnet:

$$offset = edgeCount * thickness + thickness$$

Es wird also die Anzahl der *Box-Shadows*, die die Kante bereits besitzt, mit der Dicke der Linie multipliziert und anschließend noch einmal die Dicke der Linie addiert, um nicht den Zellenrand zu überdecken. Dieser Offset muss für jede Kante aller betroffenen Zellen bestimmt werden. Die Farbe der *Box-Shadows* entspricht der Farbe des dem Coding zugeordneten Codes, wie er vom Nutzer konfiguriert wurde. Die Liste der *Box-Shadows* wird im Style-Attribut der einzelnen Zellen gespeichert und an die *ReactGrid* übergeben, die diese rendert.

Auswirkung von Änderungen der Tabellenstruktur auf die Codings

Das Verändern einer codierten Tabelle erfordert in der Regel eine Anpassung der bestehenden Codings, auf die im Folgenden eingegangen wird. Da in *CellCodings* nur eine Referenz auf einen Bereich von Zellen gespeichert ist, sind sie unabhängig vom Inhalt der Zellen und bleiben bei Bearbeitung des Zellinhhalts unverändert. Bei Änderungen der Tabellenstruktur müssen die vorhanden Codings aber angepasst werden. Wie im Kapitel 4.3.3 erläutert, kann ein Nutzer neue Zeilen und Spalten hinzufügen oder vorhandene Zeilen oder Spalten löschen. Beim Hinzufügen einer neuen Zeile oberhalb eines gesetzten Codings muss dieses um eine Zeile nach unten verschoben werden, indem *rowStart* und *rowEnd* um die Anzahl hinzugefügter Zeilen erhöht werden. Wenn eine neue Zeile innerhalb eines *CellCodings* hinzugefügt wird, dann vergrößert sich der codierte Bereich und die gespeicherte Id in *rowEnd* wird erhöht. Beim Löschen einer Zeile oberhalb eines gesetzten Codings müssen *rowStart* und *rowEnd* um die Anzahl der gelöschten Zeilen reduziert werden und beim Löschen innerhalb eines Codings wird der codierte Bereich durch das Reduzieren der Id in *rowEnd* verkleinert. Analog muss auch *columnStart* und *columnEnd* der gesetzten *CellCodings* angepasst werden,

[23] https://developer.mozilla.org/de/docs/Web/CSS/Reference/Properties/box-shadow

wenn eine neue Spalte hinzugefügt oder eine vorhandene gelöscht wird. Eine Überprüfung der Auswirkung von Änderungen der Tabellenstruktur auf die gesetzten Codings und gegebenenfalls deren Anpassung dieser wird direkt im Anschluss an die Bearbeitung durchgeführt.

Preview von *CellCodings*

Im Coding-Editor hat der Nutzer die Möglichkeiten alle gesetzten Codings eines bestimmten Codes anzuschauen. Dafür gibt es die Preview mit einer Liste an Ausschnitten aus den verschiedenen Dokumenten, die mit dem ausgewählten Code codiert wurden. Diese Preview musste erweitert werden, damit auch Ausschnitte aus den Tabellenkalkulationsdokumenten angezeigt werden, die mit dem neuen Coding-Typ *CellCoding* codiert wurden. So ein Ausschnitt ist in der Abbildung 4.10 zu sehen ist. In diesem Preview wird für jedes im Tabellenblatt gesetzte

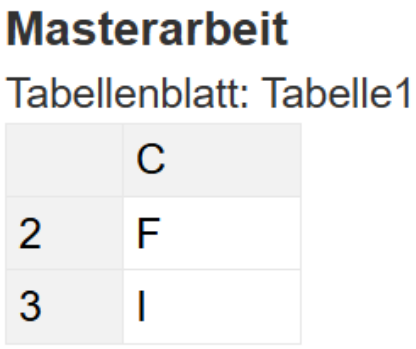


Abbildung 4.10: Preview eines codierten Bereichs in einem Tabellenkalkulationsdokument

Coding, der codierte Bereich mit der jeweiligen Zeilen- und Spaltenüberschrift abgebildet. Durch diese Überschrift kann der Nutzer erkennen, welcher Bereich innerhalb der Tabelle codiert wurde. Für die Realisierung dieser Darstellung wurde die Komponente *SectorOfTable* implementiert, die neben den Metadaten zum Laden des YDocs über die *Hocuspocus*-Verbindung, mittels Prop *boundingBox* der Bereich des Codings übergeben wird. Analog zum Tabelleneditor wird auch hierbei das im YDoc gespeicherte Tabellenkalkulationsdokument in die Datenstruktur, die von der *ReactGrid*-Komponente benötigt wird, umgeformt. Dabei werden aber nur die Zellen übergeben, die Teil der *boundingBox* sind.

Coding-Überscheidung

Der Coding-Editor von QDAcity bietet dem Nutzer im Reiter Statistik eine Übersicht zur Anzahl der Überschneidungen von Codings an. Dabei werden über alle Dokumente des Projekts hinweg die Anzahl an Überschneidungen der Codings

von 2 gewählten Codes ermittelt. Damit auch Coding-Überschneidungen für *CellCodings* erfasst werden, muss die Funktion *checkForOverlap* um die Prüfung zur Überschneidung zweier *CellCodings* erweitert werden, sodass auch *CellCodings* in der Statistik berücksichtigt werden. Neben den Zahlenwerten kann der Nutzer sich auch jede einzelne Überschneidung anschauen. Um dies auch für *CellCodings* zu ermöglichen, wurde das Enum *OverlapType* ergänzt, das die Art der Codings, die sich überschneiden, angibt. Abhängig von diesem *OverlapType* wird die Überschneidung in der neuen Komponente *PreviewCodingOverlap* unterschiedlich gerendert. Beim *OverlapType Text* wird, wie bisher, der Ausschnitt des Textes mit den betreffenden Codings gerendert und die Überschneidung mit einem Orange-Ton aus der QDAcity-Farbpalette hinterlegt. Bei Tabellen wird auf die *SectorOfTable*-Komponente zurückgegriffen, die als *boundingBox* den beide Codings umfassenden Bereich übergeben wird. Über die Prop *overlapBox* wird der Bereich der Überschneidung mitgegeben, dessen Zellen von der Komponente mit orangen Hintergrundfarbe markiert werden, die auch bei Text-Überschneidungen verwendet wird. Diese Hervorhebung ist auch in der Abbildung 4.11 zu sehen.

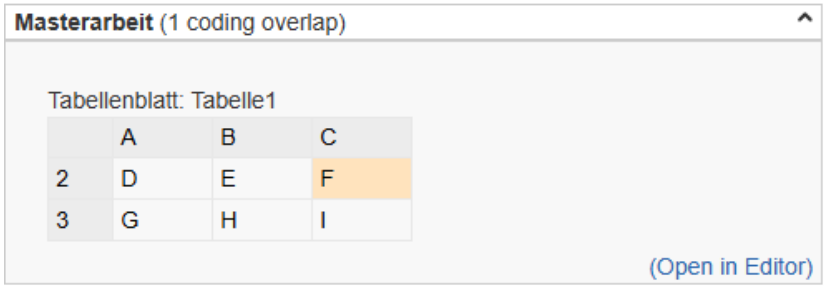


Abbildung 4.11: Ansicht einer Coding-Überschneidung in einer Tabelle

5 Evaluation

In diesem Kapitel wird evaluiert, welche der in Kapitel 2 definierten Anforderung im Rahmen der Masterarbeit erfüllt wurden. Dabei wird zunächst auf die funktionalen Anforderungen eingegangen und im Anschluss auf die nicht-funktionalen.

5.1 Evaluation der Funktionalen Anforderungen

Die funktionalen Anforderungen sind auf die beiden Bereiche Textdokumente und Tabellenkalkulationsdokumente aufgeteilt, die in den zwei folgenden Unterkapiteln bewertet werden.

5.1.1 Erweiterung der Datentypen für Textdokumente

Als erstes werden die Anforderungen für den Datentyp DOCX evaluiert und danach die für RTF-Dokumente.

Erweiterung um den Datentyp DOCX

FA 1.1.1: *Der Upload-Prozess muss dem Nutzer die Möglichkeit bieten, eine DOCX-Datei von seinem lokalen Dateisystem auszuwählen und in den Texteditor hochzuladen.*

Wie im Kapitel 4.1.1 beschrieben, wurde der Mime-Typ für DOCX-Dateien für Textdokumente freigeschaltet, wodurch der Nutzer eine DOCX-Datei hochladen kann. Mithilfe der Bibliothek *Mammoth* wird die Datei zuerst in eine HTML-Struktur und dann mit einer Konvertierungslogik zu Slate-Nodes umgewandelt. Für das Rendern der Slate-Nodes existieren die Methoden *renderLeaf* und *renderElement*, die die Textknoten und Slate-Container-Knoten zu HTML umformen (vgl Kapitel 3.2.2 und 4.1.1). Durch einen Akzeptanztest, der diese Anforderung testet, wird die Funktionalität des Uploads einer DOCX-Datei auch langfristig sichergestellt.

FA 1.1.1 ist erfüllt

FA 1.1.2: *Der Texteditor muss fähig sein, die folgenden Zeichenformatierungen, der importierten DOCX-Datei zu erkennen und als solche zu visualisieren:*

> ***FA 1.1.2.1:*** *Der Texteditor muss fähig sein, die Formatierung unterstrichen bei importierten Zeichen beizubehalten.*
>
> ***FA 1.1.2.2:*** *Der Texteditor muss fähig sein, die Formatierung fett bei importierten Zeichen beizubehalten.*
>
> ***FA 1.1.2.3:*** *Der Texteditor muss fähig sein, die Formatierung kursiv bei importierten Zeichen beizubehalten.*
>
> ***FA 1.1.2.4:*** *Der Texteditor muss fähig sein, die Formatierung hochgestellt bei importierten Zeichen beizubehalten.*
>
> ***FA 1.1.2.5:*** *Der Texteditor muss fähig sein, die Formatierung tiefgestellt bei importierten Zeichen beizubehalten.*
>
> ***FA 1.1.2.6:*** *Der Texteditor muss fähig sein, die Formatierung durchgestrichen bei importierten Zeichen beizubehalten.*

Beim Konvertieren der Dateien mit *Mammoth* werden diese Textformatierungen erkannt und die betroffenen Textabschnitte mit den entsprechenden HTML-Tags versehen. Wie im Unterpunkt Erweiterung der Zeichenformatierung erläutert, wurde die *SlateUtils*-Datei um die Konvertierung der noch fehlenden HTML-Tags erweitert, sodass die Textformatierung als Attribute der Text-Knoten in der Slate-Node-Hierarchie gespeichert werden. Außerdem wurde die *renderLeaf*-Methode um die fehlenden *switch-cases* ergänzt, damit die Textformatierung beim Rendern der Text-Knoten auf den Text anzuwenden.

FA 1.1.2 ist erfüllt

FA 1.1.3: *Der Texteditor muss fähig sein, die folgenden Strukturformatierungen, der importierten DOCX-Datei zu erkennen und als solche zu visualisieren:*

> ***FA 1.1.3.1:*** *Der Texteditor muss fähig sein, die Struktur importierter Listen beizubehalten.*
>
> ***FA 1.1.3.2:*** *Der Texteditor muss fähig sein, die Struktur importierter Tabellen beizubehalten.*
>
> ***FA 1.1.3.3:*** *Der Texteditor muss fähig sein, die Hierarchie importierter Überschriftebenen beizubehalten.*
>
> ***FA 1.1.3.4:*** *Der Texteditor muss fähig sein, die Hierarchie importierter Titel beizubehalten.*

Damit *Mammoth* importierte Titel berücksichtigt, wurde deren Zuordnung zu einem HTML-Tag bei der Konvertierung mit übergeben. Die anderen Strukturformatierungen der obigen Anforderungsliste werden standardmäßig mit den ent-

sprechenden HTML-Tags versehen. Da die Konvertierung von HTML zu Slate-Nodes aus der Verarbeitung von RTF-Dateien, die bereits Listen unterstützen, wiederverwendet wurde, waren keine weiteren Anpassungen dafür nötig. Die für die hierarchischen Überschriften inklusive Titel benötigten Anpassungen sind im Unterpunkt Hierarchische Überschriften und die für Tabellen im Unterpunkt Tabellen beschrieben. Die Strukturinformationen werden als Slate-Container-Element-Knoten persistiert und in der *renderElement*-Methode so gerendert, dass die Strukturen aus der DOCX-Datei erhalten bleiben.

FA 1.1.3 ist erfüllt

FA 1.1.4: *Der Texteditor muss fähig sein, in der DOCX-Datei eingebettete Links als solche zu visualisieren.*

Auch Links werden von *Mammoth* erkannt und mit dem entsprechenden HTML-Tag gekennzeichnet. Wie im Unterpunkt Links beschrieben, werden diese zu Slate-Nodes konvertiert und als Link gerendert. Um XSS-Attacken zu verhindern, werden nur Zieladressen, die mit *http* oder *https* beginnen, gespeichert. Andere Links verweisen mit dem Ziel # auf die geöffnete Seite.

FA 1.1.4 ist erfüllt

FA 1.1.5: *Der Texteditor muss fähig sein, Bilder aus einer importierten DOCX-Datei an der gleichen Stelle im Textfluss eingebunden anzuzeigen.*

Um diese Anforderung zu erfüllen, wurde eine Methode erstellt, die festlegt wie *Mammoth* mit den im Text eingebundenen Bilder umgehen soll. Diese Methode beinhaltet den Upload der Bilder in den GCS und das Cachen im Browser des Nutzers, was im Unterpunkt Datenübertragung und Caching-Strategie der über DOCX-Dateien hochgeladenen Bilder beschrieben ist. Die Bilder werden beim Upload der DOCX-Datei persistiert und in der HTML-Struktur an der Position innerhalb des Testflusses als Slate-Container-Element-Knoten des Typs *image* mit den Informationen Alternativtext und Id des Bildes hinzugefügt und gespeichert. Beim Rendern wird dann das Bild über einen implementierten Kontext aus dem GCS oder aus dem Cache geladen und im Textfluss angezeigt (vgl. Unterpunkt Bilder). Ein Akzeptanztest, der eine DOCX-Datei inklusive Bild hochlädt und die Position des Bildes im Texteditor von QDAcity verifiziert, schützt die Funktionalität vor Regression

FA 1.1.5 ist erfüllt

FA 1.1.6: *Im Dokument eingebettete Bilder sollten codiert werden können.*

Die im Coding-Editor für Textdokumente verwendeten *SlateTextCodings* werden mit zwei Ankerpunkten gespeichert, die die Position innerhalb des Textes festlegen. Der Offset innerhalb des Textes kann für eingebettete Bilder nicht bestimmt werden, wodurch ein Bild nicht als Ankerpunkt in Betracht kommt. Damit es

aber trotzdem Teil eines Codings sein kann, wird das Bildelement im Editor als atomares Element spezifiziert und außerdem mittels CSS-Formatierung für den Nutzer als nicht auswählbar festgelegt. Dadurch können Codings eingebettete Bilder zumindest umspannen.

FA 1.1.6 ist teilweise erfüllt

Verbesserungen für den Datentyp RTF

Bezüglich der Verbesserungen für den Datentyp RTF werden die folgenden Anforderungen evaluiert:

FA 1.2.1 *Die für den Upload der RTF-Dateien genutzte Bibliothek Tika muss auf die neue Version 3.2.3 aktualisiert werden.*

Die im Backend verwendeten Bibliotheken werden mithilfe von *maven* verwaltet. Um die *Tika*-Version zu erhöhen, musste die Versionsnummer in der Konfigurationsdatei manuell angepasst werden. Auf die von der Bibliothek verwendete Konvertierung von RTF-Dateien zu HTML-Strukturen hatte das Update auf Version 3.2.3 keine funktionellen Auswirkungen, sodass keine weiteren Anpassungen nötig waren.

FA 1.2.1 ist erfüllt

FA 1.2.2 *Die Latent Styles der RTF-Dateien sollten im Texteditor nicht als sichtbarer Text erscheinen.*

Zur Erfüllung dieser Anforderung wurde, wie im Kapitel 4.2 beschrieben, ein Algorithmus implementiert, der vor der weiteren Verarbeitung in der RTF-Struktur den *latentstyle*-Block entfernt. Dadurch wird dieser nicht mit konvertiert, persistiert und gerendert.

FA 1.2.2 ist erfüllt

5.1.2 Erweiterung der Datentypen für Tabellenkalkulationsdokumente

In diesem Kapitel werden die Anforderungen für Tabellenkalkulationsdokumente evaluiert, wobei zunächst die Anforderungen an den Tabelleneditor betrachtet werden, dann die für die Erweiterungen um den Datentyp XLSX und im Anschluss die für das Codieren der Tabellenkalkulationsdokumente.

Anforderungen an den Tabelleneditor

FA 2.1.1: *Die Anwendung muss dem Nutzer die Möglichkeit bieten, ein neues, leeres Tabellenkalkulationsdokument anzulegen.*

Die Funktionalität wurde dem Nutzer über die neue Option im Upload-Modal, das in Abbildung 4.6 zu sehen ist, zu Verfügung gestellt, die nun neben dem Erstellen eines leeren Textdokument auch die Option für ein neues Tabellenkalkulationsdokument bietet (vgl. Kapitel 4.3.6 Erstellen eines neuen Tabellenkalkulationsdokuments). Beim Ausführen dieser Funktion wird im CES ein neues YDoc mit einem einzelnen Tabellenblatt mit einer Tabelle angelegt, die aus 50 Zeilen und 10 Spalten mit Zellen ohne Wert besteht. Das erstellte Tabellenkalkulationsdokument mit dem initialen Tabellenblatt kann der Nutzer dann bearbeiten und erweitern. Durch einen Akzeptanztest wird das Erstellen eines neuen Tabellenkalkulationsdokuments nachhaltig vor Regression geschützt.

FA 2.1.1 ist erfüllt

FA 2.1.2: *Der Tabelleneditor sollte fähig sein, Tabellenkalkulationsdokumente, die mehrere Tabellenblätter enthalten, zu unterstützen, was durch die folgenden Anforderungen sichergestellt wird:*

> ***FA 2.1.2.1:*** *Eine Navigation muss dem Nutzer die Möglichkeit bieten, zwischen den Tabellenblättern zu wechseln.*
>
> ***FA 2.1.2.2:*** *Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, ein neues Tabellenblatt in einem Tabellenkalkulationsdokument hinzuzufügen.*
>
> ***FA 2.1.2.3:*** *Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, ein einzelnes Tabellenblatt aus einem Tabellenkalkulationsdokument zu löschen.*
>
> ***FA 2.1.2.4:*** *Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, den Namen der Tabellenblätter zu ändern.*

Die unterschiedlichen Tabellenblätter werden in der YMap *sheets* im YDoc gespeichert, siehe Kapitel 3.3.1. Für den Wechsel zwischen den Blättern wurde eine Reiter-Navigation implementiert, die im Kapitel 4.3.5 erläutert wird. Wie in Abbildung 4.5 zu sehen ist, bietet die Navigation im Editier-Modus auch durch einen Button neben jedem Tabellenblatt-Namen die Möglichkeit zum Löschen des entsprechenden Blattes. Über diesen Button wird, mit dem Zwischenschritt einer Zustimmung in einen Bestätigungsdialog, die kaskadenartige Löschung des zugehörigen Eintrags in der YMap *sheets* im YDoc ausgelöst. Der Plus-Button neben den Namen der Blätter löst das Erstellen eines neuen Tabellenblatts aus. Dazu wird in der YMap ein neuer Eintrag hinzugefügt, der initial eine Tabelle mit 50 Zeilen und 10 Spalten enthält. Durch jeweils einen Akzeptanztest zum Anlegen und einen zum Entfernen eines Tabellenblatts wird die Erfüllung dieser Unteranforderungen auch langfristig sichergestellt. Die Implementierung zur Umbenennung einzelner Tabellenblätter hätte zu einem größerem Refactoring-Aufwand geführt, da der Name des Blattes als Schlüssel in der YMap im YDoc verwendet wird, gegen den sich aufgrund der geringen Priorisierung der Anforderung und des zeitlich befristeten Rahmens der Masterarbeit entschieden wurde.

FA 2.1.2 ist teilweise erfüllt

FA 2.1.3: *Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, einer Tabelle sowohl eine als auch mehrere neue Zeilen und Spalten an der vom Nutzer ausgewählten Stelle hinzuzufügen.*

FA 2.1.4: *Der Tabelleneditor muss dem Nutzer die Möglichkeit bieten, von ihm ausgewählte Zeilen und Spalten zu löschen.*

Für das Hinzufügen neuer Zeilen und Spalten steht dem Nutzer zum einen das Kontext-Menü und zum anderen die Toolbar zu Verfügung. Wie im Kapitel 4.3.3 erläutert, kann über das Kontext-Menü eine Zeile unterhalb der ausgewählten Zelle und eine Spalte rechts von dieser hinzugefügt werden. Über die Toolbar kann der Nutzer zusätzlich auch Zeilen oberhalb und Spalten links der ausgewählten Zelle einfügen, und er kann auch direkt mehrere Zeilen und Spalten auf einmal hinzufügen. Die genaue Anzahl muss in diesem Fall in einem Modaldialog konkretisiert werden. Analog zum Hinzufügen von Zeilen und Spalten stehen zum Löschen einer ausgewählten Zeile oder Spalte auch entsprechende Funktionen über das Kontext-Menü und die Toolbar zur Verfügung. Die unterschiedlichen Optionen zum Einfügen und Löschen sind mit vier Akzeptanztests vollständig abgedeckt.

FA 2.1.3 und FA 2.1.4 sind erfüllt

FA 2.1.5: *Die Tabellen sollten eine Zeilen- und Spaltenüberschrift haben.*

Um diese Anforderung zu erfüllen, wird bei der Umformung in der Methode *yDocToReactGrid*, die die Tabelle aus der Speicherstruktur aus dem YDoc in die von der *ReactGrid* benötigten Struktur umwandelt, spezielle Zellen hinzugefügt (vgl. Kapitel 4.3.2). Diese Zellen unterscheiden sich durch dem Zell-Typ *Header* sowohl farblich als auch dadurch, dass man sie weder auswählen noch bearbeiten kann, von normalen Zellen der Tabelle. Die Überschriftzellen der Spalten erhalten eine alphabetische Indexierung und die der Zeilen eine numerische, wie sie in Abbildung 4.1 zu sehen sind.

FA 2.1.5 ist erfüllt

FA 2.1.6: *Der Tabelleneditor sollte fähig sein, Formeln innerhalb von Zellen auszuwerten.*

Weil es keine für den kommerziellen Gebrauch kostenlose Javascript-Bibliothek gibt, die auch Formeln nativ unterstützt, hätte die Integration dieser Funktionalität erheblichen Mehraufwand verursacht. Da der Fokus von QDAcity auf dem Codieren und weniger auf dem Bearbeiten von Dokumenten liegt, wurde diese Anforderung niedriger priorisiert und konnte letztendlich aufgrund der beschränkten Zeit einer Masterarbeit nicht mehr umgesetzt werden.

FA 2.1.6 ist nicht erfüllt

FA 2.1.7: *Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, durch Synchronisation für mehrere Nutzer, kollaboratives Arbeiten zu ermöglichen.*

Zur Erfüllung dieser Anforderung wurde eine Speicherstruktur des Tabellenkalkulationsdokuments mit Datentypen von *Yjs* innerhalb eines YDocs gewählt, die in Abbildung 3.7 visualisiert wird. Bei dieser für das kollaborative Arbeiten konzipierten Datenstruktur werden Änderungen am Dokument mithilfe von Transaktionen durchgeführt. Zur Synchronisation der YDocs für mehrere Nutzer wird wie bei Textdokumenten der CES genutzt, der hierfür auf der *Hocuspocus*-Technologie aufbaut, siehe Kapitel 4.3.1

FA 2.1.7 ist erfüllt

FA 2.1.8: *Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, den Text innerhalb einer Zelle umbrechen zu lassen und manuelle Zeilenumbrüche einzufügen, damit auch längerer Text in einer Zelle angezeigt werden kann.*

Der Nutzer kann das standardmäßige Verhalten, dass der Text am Rand der Zelle abgeschnitten wird, sowohl über die Toolbar als auch über das Kontext-Menü zu einem Textumbruch ändern. Da die Zellen der Tabelle dem implementierten Zell-Typ *multiline-text* zugeordnet sind, der als Eingabefeld *textareas* verwendet, werden manuelle Zeilenumbrüche durch die Eingabe der Enter-Taste in Kombination mit Steuerung, Alt oder Shift unterstützt. Damit alle Zeilen der Zellen bei aktiviertem Textumbruch angezeigt werden, wird die dafür notwendige Höhe der Zeile bestimmt und an die *ReactGrid*-Komponente übergeben (vgl. Kapitel 4.3.4). Das Wechseln des Textumbruchverhaltens wird mit zwei Akzeptanztests vor einer Regression geschützt.

FA 2.1.8 ist erfüllt

FA 2.1.9: *Der Tabelleneditor sollte dem Nutzer die Möglichkeit bieten, die Breite einer Spalte zu ändern.*

Wie im Kapitel 4.3.2 erläutert, bietet die *ReactGrid*-Komponente die Option, Spalten mit dem Attribut *resizable* zu spezifizieren, was dem Nutzer ermöglicht, durch Verschieben der Spaltenbegrenzung, die Breite der Spalten zu ändern. Diese Funktion ist bei allen Spalten, außer der Überschriftspalte, im Tabelleneditor freigeschaltet. Um die Breite zu persistieren und dadruch beim Neuladen ein Zurücksetzen auf die Standartbreite zu verhindern, wird die Breite der Spalte in den dazugehörigen Zellen innerhalb ihrer YMap gespeichert. Bei der Umformung zu den für die *React-Grid*-Komponente benötigten Props innerhalb der Methode *yDocToReactGrid* wird die hinterlegte Breite der Spalte ausgelesen und mit übergeben.

FA 2.1.9 ist erfüllt

Anforderungen für die Erweiterung um den Datentyp XLSX

FA 2.2.1: *Der Upload-Prozess muss dem Nutzer die Möglichkeit bieten, eine XLSX-Datei auszuwählen und in den Tabelleneditor hochzuladen.*

Zur Erfüllung dieser Anforderung wurde der Mime-Typ für XLSX-Dateien für Tabellenkalkulationsdokumente freigeschaltet, indem bei der Methode *getSupportetMimeTypes* für die Klasse *SpreadsheetDocument* der zurück gegebene Mime-Typ *application/vnd. openxmlformats-officedocument.spreadsheetml.sheet* hinterlegt wurde. Dadurch kann der Nutzer im Upload-Modal eine XLSX-Datei von seinem lokalen Dateisystem auswählen und hochladen, siehe Kapitel 3.3.1. Durch einen Upload einer XLSX-Datei in einem Akzeptanztest wird die Funktionalität auch langfristig sichergestellt.

FA 2.2.1 ist erfüllt

FA 2.2.2: *Der Tabelleneditor muss fähig sein, die tabellarische Struktur mit den Zellwerten aus einem als XLSX-Datei hochgeladenen Tabellenkalkulationsdokument zu übernehmen.*

Für diese Funktionalität werden die Zellwerte der hochgeladenen XLSX-Datei mithilfe der Bibliothek *exceljs* ausgelesen und in die Speicherstruktur der Tabellenkalkulationsdokumente in QDAcity umgeformt, die im Kapitel 3.3.1 vorgestellt wird. Das dabei entstehende YDoc kann dann analog zu den direkt in QDAcity erstellten Tabellenkalkulationsdokumenten bearbeitet und codiert werden (vgl. Kapitel 4.3.6 Upload einer XLSX-Datei). Das Übernehmen der korrekten Werte, auch bei Formeln in der ursprünglichen XLSX-Datei, wird durch einen Akzeptanztest verifiziert.

FA 2.2.2 ist erfüllt

Anforderungen für das Codieren von Tabellen

FA 2.3.1: *Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, ausgewählte Zellen in einer Tabelle zu codieren.*

Damit der Nutzer einen Bereich an Zellen innerhalb einer Tabelle markieren kann, wurde im *ReactGrid* die Prop *enableRangeSelection* auf *true* gesetzt. Die ausgewählten Zellen werden im *State selectedRanges* gespeichert. Wenn ein neues Coding gesetzt wird, wird aus diesem *State* ein *CellCoding* für den Bereich erstellt. Codings vom gleichen Code, die innerhalb dieses Bereichs liegen, werden gelöscht. Die neuen Codings werden analog zu den anderen Coding-Typen im YDoc persistiert (vgl. Kapitel 4.3.7 Neues Coding hinzufügen). Auch das Setzen eines solchen *CellCodings* ist durch einen Akzeptanztest abgedeckt.

FA 2.3.1 ist erfüllt

FA 2.3.2: *Der Coding-Editor muss fähig sein, gesetzte Codings der geöffneten Tabelle auf den beiden folgenden Arten anzuzeigen:*

> ***FA 2.3.2.1:*** *Der Coding-Editor muss fähig sein, gesetzte Codings in Coding-Brackets links neben der Datei anzuzeigen.*
>
> ***FA 2.3.2.2:*** *Der Coding-Editor muss fähig sein, den Bereich eines gesetzten Codings innerhalb der geöffneten Tabelle durch eine Umrahmung anzuzeigen.*

Für die Visualisierung gesetzter Codings wurden die Optionen zur Darstellung mit *Coding-Brackets* und innerhalb der Tabelle implementiert, zwischen den der Nutzer in einem Menü mit Checkboxen wählen kann, welches in Abbildung 4.9 zu sehen ist. Die Kombination dieser beiden Optionen ist in Abbildung 4.8 zu sehen. Für die *Coding-Brackets* wird die *CodingBrackets*-Komponente wiederverwendet, deren Höhe und Offset für jedes Coding durch Auslesen der DOM-Werte bestimmt werden muss. Für die Visualisierung innerhalb der Tabelle wurde sich für eine Umrahmung mittels *Box-Shadows* entschieden. Da einzelne Zellen die Ränder mehrerer Codings sein können, werden die *Box-Shadows* mit einem individuell für jedes dieser Codings berechneten Offset gesetzt. Als Farbe der Schatten wird die vom Nutzer konfigurierte Farbe des zugehörigen Codes verwendet, siehe Kapitel 4.3.7 Visuelle Darstellung von Codings. Durch einen Akzeptanztest sind beide Visualisierungsarten vor einer Regression geschützt.

FA 2.3.2 ist erfüllt

FA 2.3.3: *Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, ausgewählte Zellen, wenn sie Bestandteil eines Codings sind, aus diesem Coding zu entfernen. Wenn alle Zellen eines Codings ausgewählt sind, muss dieses Coding gelöscht werden.*

Durch die Definition der *CellCodings* im Kapitel 3.3.4 wird festgelegt, dass sie nur rechteckige Bereiche innerhalb einer Tabelle codieren können. Um diese Anforderung dennoch zu erfüllen, wurden im Kapitel 4.3.7 Bereich von Coding entfernen abhängig vom zu entfernenden Bereich 6 unterschiedliche Fälle definiert. Je nach Fall wird das vom Entfernen betroffene Coding aufgesplittet oder gelöscht, um den verbleibenden codierten Teil vollständig abzudecken.

FA 2.3.3 ist erfüllt

FA 2.3.4: *Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, das Coding-Bracket eines gesetzten Codings anzuklicken, um dessen codierte Zellen in der Tabelle visuell hervorzuheben.*

Zur Erfüllung dieser Anforderung wurde die Methode *activateCodingInEditor* implementiert, die ausgeführt wird, wenn eine *Coding-Bracket* angeklickt wird. Diese setzt den *State highlightedCells* auf ein Array mit den codierten Zellen des ausgewählten Codings, das der *ReactGrid*-Komponente übergeben wird. Dadurch

werden diese Zellen blau umrandet (vgl. Kapitel 4.3.7 Visuelle Darstellung von Codings).

FA 2.3.4 ist erfüllt

FA 2.3.5: *Der Coding-Editor muss fähig sein, beim Hinzufügen von Spalten und Zeilen innerhalb eines codierten Bereichs, dieses Coding entsprechend um die hinzugefügten Zellen zu erweitern.*

FA 2.3.6: *Der Coding-Editor muss fähig sein, beim Löschen von Spalten und Zeilen innerhalb eines Codings, dieses Coding entsprechend um die entfernten Zellen zu reduzieren.*

Wie im Kapitel 4.3.7 Auswirkung von Änderungen der Tabellenstruktur auf die Codings beschrieben, werden beim Ändern der Tabellenstruktur die Auswirkungen auf die innerhalb der Tabelle gesetzten *CellCodings* geprüft. Dabei werden die Codings durch Anpassung der Attribute *rowStart*, *rowEnd*, *columnStart* und *columnEnd* entsprechend der vorgenommenen Änderung verschoben. Bei einer Änderung innerhalb eines Codings vergrößert beziehungsweise verkleinert sich der codierte Bereich.

FA 2.3.5 und FA 2.3.6 sind erfüllt

FA 2.3.7 *Das System sollte fähig sein, Tabellenkalkulationsdokumente bei der Bestimmung des Intercoder Agreements zu berücksichtigen.*

Diese Anforderung konnte aufgrund der gleichzeitig stattgefundenen Umstellung auf einen neuen Service zur Berechnung des *Intercoder Agreements* nicht implementiert werden. Da diese weiterhin nur auf Basis der Textdokumente durchgeführt wird, erhält der Nutzer beim Erstellen eines Berichtes zum *Intercoder Agreement* einen Hinweis, wenn das Projekt nicht unterstützte Dokumenten enthält.

FA 2.3.7 ist nicht erfüllt

FA 2.3.8: *Der Coding-Editor muss dem Nutzer die Möglichkeit bieten, die mit einem bestimmten Code codierten Zellen in einer Liste der Codings diesen Codes zu sehen.*

Wie im Kapitel 4.3.7 Preview von CellCodings beschrieben, werden die *CellCodings* in der Liste eines bestimmten Codes mit Hilfe der *SectorOfTable*-Komponente visualisiert. Dabei wird jeweils nur der codierte Bereich der Tabelle inklusive der zugehörigen Zeilen- & Spaltenüberschrift gerendert, was in Abbildung 4.10 zu sehen ist und in einem Akzeptanztest verifiziert wird.

FA 2.3.8 ist erfüllt

FA 2.3.9: *Der Coding-Editor muss fähig sein, Überschneidungen zweier Codings innerhalb eines Tabellenblattes in der Statistik zu berücksichtigen und diese zu visualisieren.*

Die zur Erfüllung dieser Anforderung nötigen Implementierungen sind im Kapitel 4.3.7 Coding-Überscheidung beschrieben. Um Überschneidungen bei *CellCodings* in der Statistik zu berücksichtigen, wurde die Methode *checkForOverlap* um die Prüfung der Überschneidung zweier *CellCodings* erweitert. Diese werden mit der *SectorOfTable*-Komponente visualisiert, die den Ausschnitt der Tabelle der betroffenen Zellen rendert. Die Zellen, bei denen sich die Codings überschneiden, werden orange hinterlegt. Durch einen Akzeptanztest wird dies vor Regression geschützt.

FA 2.3.9 ist erfüllt

5.2 Evaluation der Nicht-Funktionalen Anforderungen

Die Evaluation der Nicht-Funktionalen Anforderungen in diesem Kapitel ist auf die Bereiche Performanz und Effizienz, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit, Wartbarkeit und Übertragbarkeit des ISO-Standards 25010 aufgeteilt.

5.2.1 Performanz und Effizienz

NFA 1.1: *Der Inhalt der DOCX-Datei muss im Frontend in einen Blob umgewandelt und in den GCS hochgeladen werden, ohne ihn an das Backend zu schicken.*

Wie im Kapitel 3.2.2 beschrieben, wird die hochgeladene DOCX-Datei im Frontend zu Slate Nodes konvertiert, die in ein leeres YDoc geladen werden. Diese Änderung am YDoc wird dann direkt aus dem Frontend in den GCS übertragen. Dadurch wird die Konvertierung im Browser des Nutzers ausgeführt und führt zu keiner zusätzlichen Last im Backend, das in der GCP läuft und begrenzte Rechenkapazität hat.

NFA 1.1 ist erfüllt

NFA 1.2: *Der Inhalt der RTF-Datei sollte im Frontend in einen Blob umgewandelt und in den GCS hochgeladen werden, ohne ihn an das Backend zu schicken.*

Zur Erfüllung dieser Anforderung wurde eine Javascript Bibliothek gesucht, die die Java Bibliothek Tika im Backend ersetzt und die Umformung von RTF zu HTML im Frontend durchführt. Es konnte allerdings keine Bibliothek mit äquiva-

lenter Funktionalität, gefunden werden, die die Ansprüche des Projektes erfüllt und weiterhin Listen innerhalb von RTFs unterstützt. Eine eigenständige Programmierung der Konvertierungsfunktion, ohne Zurückgreifen auf eine etablierte Bibliothek, hätte zu erheblichem Mehraufwand geführt, ohne einen direkten Mehrwert für den Nutzer zu schaffen. Deshalb wurde die Anforderung nicht umgesetzt.

NFA 1.2 ist nicht erfüllt

NFA 1.3: *Der Inhalt der XLSX-Datei muss im Frontend in einen Blob umgewandelt und in den GCS hochgeladen werden, ohne ihn an das Backend zu schicken.*

Durch die Bibliothek *exceljs* ist es möglich, die XLSX-Datei im Frontend auszulesen und in die Speicherstruktur von Tabellenkalkulationsdokumenten innerhalb eines YDocs umzuformen. Die Änderungen am YDoc werden dann ohne den Umweg über das Backend in dem GCS persistiert, wodurch keine Rechenlast des Backends für die Konvertierung beansprucht wird (vgl. Kapitel 4.3.6 Upload einer XLSX-Datei.

NFA 1.3 ist erfüllt

NFA 1.4: *Ein mit der DOCX-Datei hochgeladenes Bild muss im Frontend gecacht werden, um den Datenverkehr zum GCS gering zu halten.*

Wie im Kapitel 3.2.3 Datenübertragung und Caching-Strategie der über DOCX-Dateien hochgeladenen Bilder vorgestellt, werden alle Bilder direkt beim Upload der DOCX-Datei in der *IndexedDB imageStorage* im lokalen Speicher des Browsers des Nutzers gespeichert. Wenn ein Bild aus einer DOCX-Datei aus dem GCS geladen werden muss, das sich nicht mehr im Cache befindet, wird dieses direkt wieder der *IndexedDB* hinzugefügt, um erneute Zugriffe auf den GCS zu vermeiden.

NFA 1.4 ist erfüllt

5.2.2 Kompatibilität

NFA 2.1: *Die Implementierung sollte für eine einheitliche Architektur auf die bereits verwendeten Technologien zurückgreifen.*

Die für die neuen Funktionen notwendigen Implementierungen wurden im Frontend mithilfe des vorhandenen React-Frameworks und im Backend mittels Java vorgenommen. Für die Unterstützung des kollaborativen Arbeitens wurde die bereits eingesetzten Technologien von Yjs und Hocuspocus genutzt.

NFA 2.1 ist erfüllt

5.2.3 Benutzbarkeit

NFA 3.1: *Die Benutzeroberfläche muss die Sprachen Deutsch und Englisch unterstützen.*

Mithilfe von *FormattedMessages* der Bibliothek *react-intl*[1] wird der Text der Benutzeroberfläche automatisch übersetzt. Dafür wird als *defaultMessage* der englische Text und eine Id der Komponente *FormattedMessage* übergeben. Die deutsche Übersetzung mit der Id der Nachricht wird in der Datei *core.de.txt* definiert.

NFA 3.1 ist erfüllt

NFA 3.2: *Die Benutzeroberfläche sollte die WCAG 2.1 der Stufe AA erfüllen.*

Die Barrierefreiheit der Benutzeroberfläche wurde mit der Chrome-Erweiterung *Lighthouse*[2] überprüft. Dabei stellte sich unter anderem heraus, dass mehrere Elemente der bisherigen Implementierung bereits einen zu geringen Farbkontrast haben. Beispielweise hat der Button zum Wechsel in den Editier-Modus, der aus einer weißen Schrift mit dem Farbcode #ffffff auf blauen Hintergrund mit dem Farbcode #5B94D9 besteht, nur einen Kontrast von 3.1:1.[3] Für die Schriftgröße im Button müsste diese bei mindestens 4.5:1 liegen, um den Regeln der WCAG 2.1 der Stufe AA zu erfüllen.[4] Durch diesen Verstoß erfüllt bereits die bisherige Version von QDAcity die WCAG 2.1 der Stufe AA nicht. Eine Korrektur dieser Regelverstöße wurde nicht im Rahmen der Masterarbeit durchgeführt.

NFA 3.2 ist nicht erfüllt

NFA 3.3: *Die Benutzeroberfläche müssen dem QDAcity Designsystem entsprechen*

Zur Erfüllung dieser Anforderung wurden die Konventionen für UI-Komponenten, Layouts und Schriftarten des Projekts eingehalten. Außerdem wurde nur auf die Farben der Farbpalette von QDAcity zurückgegriffen.

NFA 3.3 ist erfüllt

NFA 3.4: *Bei der Implementierung von Buttons sollte auf wiederverwendbare Komponenten zurückgegriffen werden.*

Auch wenn für die Buttons in der Toolbar die Komponente *BtnDefault* wiederverwendet wurde, wurde diese Anforderung nicht erfüllt. Denn die Buttons für die Navigation zwischen den Tabellenblättern und zum Anlegen und Löschen eines Blattes bauen nicht auf die vorhandenen UI-Komponenten auf, sondern wurden

[1] https://github.com/formatjs/formatjs

[2] https://chromewebstore.google.com/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk?hl=de&pli=1

[3] https://barrierefreies.design/werkzeuge/kontrastrechner-fuer-farben#ffffff-5b94d9

[4] https://www.w3.org/WAI/WCAG21/Understanding/contrast-minimum.html

individuell für diesen Verwendungszweck implementiert.

NFA 3.4 ist nicht erfüllt

5.2.4 Zuverlässigkeit

NFA 4.1: *Der Rollout einer neuen Funktion muss über ein Feature-Flag im lokalen Speicher des Browsers für den normalen Nutzer verborgen sein, bis die Funktionalität für alle freigegeben ist.*

Bis die Unterstützung des Dateiformats DOCX und XLSX vollständig implementiert und getestet war, wurden die neuen Mime-Typen nur über einen geschützten Endpunkt erlaubt, der von der Anwendung nur verwendet wurde, wenn das entsprechende Feature-Flag im lokalen Speicher des Browsers gesetzt war. Diese Absicherung verhindert, dass reguläre Nutzer der Software DOCX- oder XLSX-Dateien hochladen oder neue Tabellen erstellen können, bevor diese Funktionalitäten vollständig umgesetzt sind. Nach Fertigstellung und erfolgreichem Test wurde der neue Mime-Typ für DOCX-Dateien als erlaubter Mime-Typ für Textdokumente und der Mime-Typ für XLSX-Dateien als erlaubter Mime-Typ für Tabellenkalkulationsdokumente auch über den regulären Endpunkt *getSupportedDocumentTypes* zurückgegeben.

NFA 4.1 ist erfüllt

NFA 4.2: *Es muss sichergestellt werden, dass bereits vorhandene Funktionalität durch Änderungen nicht gestört wird.*

Die Auswirkungen von Änderungen auf vorhandene Funktionalität wird durch die Ausführung der Akzeptanztest bei jedem Pipeline-Durchlauf getestet. Die Merge Requests mit der Implementierung wurden erst nach erfolgreichem Durchlauf der Pipeline gemergt.

NFA 4.2 ist erfüllt

5.2.5 Sicherheit

NFA 5.1: *Nur Nutzer mit Leseberechtigung für die Datei dürfen in der Lage sein, ein Bild, das beim Upload-Prozess einer DOCX-Datei in den GCS hochgeladenen wird, herunterzuladen.*

Zur Erfüllung dieser Anforderung wird beim Aufruf des Endpunkts *getSignedDownloadImageUrlsForTextDocumentId* in der Klasse *DocumentLegacyEndpoint* die Leseberechtigung für die Datei geprüft, bevor eine signierte URL zum Download des Bildes aus dem GCS zurückgeliefert wird. Auch beim Aufruf des Endpunkts *getSignedDownloadImageUrlsForTextDocumentIdForReview* wird überprüft, ob der Nutzer berechtigt ist, ein Review für diese Datei durchzuführen, und nur

dann die Download-URL zurückgegeben. Mit JUnit-Tests wird sichergestellt, dass die Endpunkte auch langfristig nur den autorisierten Nutzern eine entsprechende Download-URL zu Verfügung stellen.

NFA 5.1 ist erfüllt

NFA 5.2: *Beim Upload-Prozess einer DOCX-Datei müssen XSS-Attacken verhindert werden.*

Das Einschleusen von XSS-Attacken durch eingebettete Links im DOCX-Dateien wird durch ein Filtern der Link-Ziele verhindert, das ausschließlich Links erlaubt, die mit *http* oder *https* beginnen. Wenn ein Link mit einem anderen Ziel, wie zum Beispiel mit einem Javascript-Befehl, hochgeladen wird, dann wird das Link-Ziel mit # überschrieben und persistiert, sodass der Link auf das aktuell geöffnete Dokument zeigt.

NFA 5.2 ist erfüllt

NFA 5.3: *Bei allen Requests an das Backend muss der Nutzer authentifiziert werden.*

Zur Erfüllung dieser Anforderung wird beim Aufruf der neuen Endpunkte *getSignedImageUploadUrlForTextDocumentId*, *getSignedDownloadImageUrlsForTextDocumentId* und *getSignedDownloadImageUrlsForTextDocumentIdForReview* analog zu den bereits vorhandenen Endpunkten im Projekt die Authentifizierung mit einem Bearer-Token verifiziert.

NFA 5.3 ist erfüllt

NFA 5.4: *Die Daten müssen verschlüsselt in der Datenbank gespeichert werden.*

Die Daten werden ausschließlich im GCS persistiert, der die Daten serverseitig verschlüsselt.[5]

NFA 5.4 ist erfüllt

NFA 5.5: *Die in QDAcity verwendete rollen-basierte Zugriffskontrolle sollte verwendet werden, um bei Projektteilnehmern zwischen lesendem und schreibendem Zugriff unterscheiden zu können.*

Sowohl bei den neuen Endpunkten im Backend als auch bei der Prüfung von Lese- und Schreibberechtigung auf das YDoc im CES mittels *SpreadsheetDocHandler* wurde auf die rollen-basierte Zugriffskontrolle zurückgegriffen.

NFA 5.5 ist erfüllt

[5] https://docs.cloud.google.com/storage/docs/encryption/default-keys?hl=de

5.2.6 Wartbarkeit

NFA 6.1: *Um die HTML-zu-Slate-Node-Umwandlung aus dem CES im Frontend wiederzuverwenden, dürfen die benötigten Methoden manuell kopiert werden. Für eine einfachere Wartung sollten die Kopien der Methoden im Frontend und die Originale im CES identisch und das Vorgehen dokumentiert sein.*

Zur Erfüllung der Anforderung NFA 1.1 war es notwendig die Umwandlung aus dem CES ins Frontend zu kopieren. Damit diese Anforderung erfüllt ist, wurden Kopien und Originale identisch implementiert. Aufgrund der unterschiedlichen Laufzeitumgebungen wurde dafür eine Fallunterscheidung implementiert, die sicherstellt, dass der Algorithmus sowohl im Browser als auch in der Node.js-Umgebung des CES ausführbar ist. Mithilfe von Code-Kommentaren wird in beiden Instanzen auf das Kopieren hingewiesen.

NFA 6.1 ist erfüllt

NFA 6.2: *Alle neuen Endpunkte im Backend sollten eine Codezeilentestabdeckung mit Unit-Tests von 100% haben.*

Im Rahmen der Masterarbeit wurden drei neue Endpunkte implementiert, deren Funktionalität mit JUnit-Tests geprüft werden. Im Anhang A wird die Codezeilentestabdeckung, die mit *Jacoco*[6] gemessen wurde, näher betrachtet. Der von *Jacoco* erstellte Report bestätigt eine Codezeilentestabdeckung der neuen Endpunkte von 100%.

NFA 6.2 ist erfüllt

NFA 6.3: *Alle neuen Funktionalitäten müssen mit Akzeptanztests geprüft werden.*

Um die neuen Funktionalitäten mit Akzeptanztests zu prüfen, wurde ein Test implementiert, der den Upload und die Anzeige von DOCX-Dateien mit eingebetteten Bildern, Links und Strukturformatierungen prüft. Außerdem wurden 11 neue Akzeptanztests geschrieben, die den Tabelleneditor, den Upload einer XLSX-Datei, das Erstellen eines neuen Tabellenkalkulationsdokuments und das Codieren mit *CellCodings* testen. Ein zusätzlicher Akzeptanztest prüft, ob Tabellenkalkulationsdokumente im Validierungsprojekt angezeigt und codiert werden können.

NFA 6.3 ist erfüllt

NFA 6.4: *Alle Java-Methoden, deren Umfang über eine einzeilige Implementierung hinausgeht, sollten mit JavaDoc-Kommentaren dokumentiert werden.*

Alle im Rahmen der Arbeit eingeführten Java-Methoden, die länger als eine Zeile

[6] https://www.jacoco.org/jacoco/

sind, erhielten eine Dokumentation mit *JavaDoc*-Kommentaren, damit der Code für andere Entwickler nachvollziehbar ist.

NFA 6.4 ist erfüllt

NFA 6.5: *Das Prinzip der Trennung von Verantwortlichkeiten sollte eingehalten werden.*

Das Prinzip wird eingehalten, da darauf geachtet wurde, dass die einzelnen Komponenten jeweils klar getrennte Aufgaben übernehmen. Neben der Aufteilung in verschiedene Komponenten wurden die Algorithmen für das Laden der Bilder einer hochgeladenen DOCX-Datei und für das Bearbeiten der Tabellenstrukturen auf separate Kontexte aufgeteilt.

NFA 6.5 ist erfüllt

6 Zukünftige Arbeiten

Die bereits umgesetzten Anforderungen lassen Potentiale für weitere Verbesserungen erkennen, die in diesem Kapitel vorgestellt werden. Ein Ziel sollte sein, die beim Upload von DOCX-Dateien übernommenen Formatierungen und eingebetteten Inhalte auch im Texteditor zu unterstützen. Dadurch können diese bei allen Textdokumenten, auch den direkt in QDAcity erstellten, verwendet werden. Im Rahmen der Masterarbeit wurde die Unterstützung für hochgestellte, tiefgestellte und durchgestrichene Zeichen in Textdokumenten implementiert. Diese Zeichenformatierungen werden von hochgeladenen DOCX-Dateien übernommen, können aber nicht im Texteditor von QDAcity gesetzt werden. Damit auch Zeichen im Editor entsprechend formatiert werden können, sollte die bereits vorhandene Toolbar mit Buttons für die zusätzlichen Zeichenformatierungen erweitert werden. Außerdem sollte es einen Button zum Hochladen von Bildern geben. Die hinterlegte Funktion kann dann intern auf die Implementierung zum Upload und Download der Bilder, die in dieser Masterarbeit vorgestellt wurden, zurückgreifen. Auch die Unterstützung der Links kann verbessert werden, indem eine Funktion zum nachträglichen Ändern im Texteditor hinzugefügt wird. Dabei sollte man den Text des Links und die Ziel-URL unabhängig voneinander bearbeiten und auch neue Links erstellen können.
Auch die Unterstützung von Tabellenkalkulationsdokumenten kann noch verbessert werden. Sobald der Umbau auf den neuen Service zur Bestimmung des *Intercoder Agreements* fertig gestellt ist, muss dieser erweitert werden, um auch *CellCodings* zu berücksichtigen. Im Rahmen der Masterarbeit wurde der Upload von XLSX-Dateien realisiert. Eine Erweiterung für den Upload von CSV-Dateien und weiteren Datentypen für Tabellenkalkulationsdokumenten würde die Kompatibilität weiter steigern. Eine Konvertierung durch den Nutzer vor dem Upload würde entfallen. Dazu bräuchte man nur eine initiale Umwandlung der Datei in die im Kapitel 3.3.1 vorgestellte Speicherstrukturen für Tabellenkalkulationsdokumente. Die implementierten Prozesse würden dann auch für diesen Datentyp greifen. Eine weitere Verbesserung wäre die Unterstützung von Formeln innerhalb von Tabellenkalkulationsdokumenten, die aufgrund der zeitlichen Beschränkung der Arbeit nicht mehr realisiert werden konnte. Durch den Einsatz einer Java-

script Bibliothek wie formulajs[1] könnte der Aufwand bei der Implementierung reduziert werden.

[1] https://github.com/formulajs/formulajs

7 Fazit

Im Rahmen der Arbeit wurde das Ziel, die effektive Unterstützung von Datentriangulaion in Forschungsprojekten in QDAcity zu verbessern, vollständig erreicht. Mit DOCX-Dateien kann QDAcity jetzt den am weitesten verbreiteten Datentyp für Textdokumente verarbeiten, was die Benutzerfreundlichkeit erheblich erhöht, da diese Dokumente nun nahtlos importiert und im Texteditor von QDAcity weiterverarbeitet und analysiert werden können. Bei diesem Import bleibt auch die Struktur von komplexen DOCX-Dateien mit Zeichen- und Strukturformatierungen, eingebetteten Links, Bildern und Tabellen erhalten. Die dafür notwendigen Erweiterung des Texteditors sind eine generelle Verbesserung für das Arbeiten mit Textdokumenten innerhalb von QDAcity, weil auf die mit dieser Arbeit gelegten Grundlagen zum Persistieren und Anzeigen komplexer Textformate aufgebaut werden kann, um diese erweiterten Formatierungen dem Nutzer generell für die Arbeit im Texteditor zur Verfügung zu stellen.
Die Unterstützung von RTF-Dateien wurde im Rahmen der Arbeit verbessert, indem ein von Nutzern der Anwendung gemeldetes Problem behoben wurde. Die von Textverarbeitungsprogrammen in die RTF-Struktur eingefügten Formatvorlagen werden nun nicht mehr als Text im Texteditor von QDAcity angezeigt, sodass sich der Nutzer bei der Arbeit mit solchen Dokumenten ausschließlich auf den Textinhalt konzentrieren kann.
Mit der Realisierung der Unterstützung von Tabellenkalkulationsdokumenten wurde ein großer Sprung im Hinblick auf die Nutzbarkeit von QDAcity für Projekte mit unterschiedlichen Datenquellen erreicht. Dafür wurde ein Tabelleneditor implementiert, der dem Nutzer nahezu den kompletten Funktionsumfang konventioneller Tabellenkalkulationssoftware bietet, ohne auf die für QDAcity charakteristische Möglichkeit der kollaborativen Zusammenarbeit verzichten zu müssen. Mit der durchgängigen Verwendung von CRDTs für das Persistieren von Tabellenkalkulationsdokumenten können Änderungen konfliktfrei synchronisiert werden. Es wurde außerdem sichergestellt, dass auch für Qualitative Datenanalyse übliche Tabellen mit großen Textinhalten in einzelnen Zellen benutzerfreundlich und übersichtlich verarbeitet werden können.
Zum Codieren der Tabellenkalkulationsdokumente wurde ein neuer Coding-Typ eingeführt, der das Codieren von einzelnen Zellen und Zellbereichen ermöglicht.

Tabellenkalkulationsdokumente können entweder direkt in QDAcity erstellt oder durch Hochladen einer XLSX-Datei integriert werden. Die Unterstützung dieses weit verbreiteten Datentyps für Tabellenkalkulationsdokumente ermöglicht einen nahtlosen Übergang von konventioneller Tabellenkalkulationssoftware zu QDAcity und ist deshalb eine erhebliche Verbesserung der Nutzerfreundlichkeit.
Der Erfolg der Arbeit zeigt sich unter anderem in der Erfüllung der anfänglich definierten Anforderung. Alle 19 funktionalen und 12 nicht-funktionale Anforderungen, die als *Muss*-Anforderungen priorisiert waren, wurden im Rahmen der Masterarbeit erfüllt. Zusätzlich wurden neben 7 nicht-funktionalen auch 5 funktionale Anforderungen vollständig und 2 teilweise umgesetzt, die als Wunsch-Anforderungen definiert waren. Im Zuge der konkreten Umsetzung hat sich gezeigt, dass die 5 übrigen Soll-Anforderungen nicht im vollen Umfang umsetzbar waren, da sie einen Mehraufwand bedeutet hätten, der erheblich über den zeitlichen Rahmen der Masterarbeit hinaus gegangen wäre.
Mit Abschluss der Masterarbeit sind bereits alle neuen Funktionalitäten integriert und können in dem Produktivsystem, das über den Link https://qdacity.com/de/ öffentlich erreichbar ist, verwendet werden. Bei der Implementierung wurde ein großer Fokus auf Wartbarkeit und eine umfangreiche Testabdeckung gelegt, um unbemerkte Regression von Funktionalitäten zu verhindern und auch weniger benutzte Funktionen langfristig vor Fehlern zu schützten. Die zukünftige Weiterentwicklung der Software wird dadurch vereinfacht und Nutzer profitieren dauerhaft von der erweiterten Funktionalität zur Unterstützung von DOCX- und XLSX-Dateien.

Anhänge

A Codezeilentestabdeckung der neuen Endpunkt

Im folgenden wird die Codezeilentestabdeckung der neuen Endpunkte, die im Rahmen der Masterarbeit implementiert wurden, betrachtet. Die Testabdeckung wurde mit Jacoco bestimmt, aus dessen Report die Abbildungen 1 bis 4 entnommen sind und in denen die betroffenen Methoden rot umrandet sind. Die beiden neuen Endpunkte *getSignedImageUploadUrlForTextDocumentId* und *getSignedDownloadImageUrlsForTextDocumentId* der Klasse *DocumentLegacyEnpoint* haben, wie in Abbildung 1 zu sehen, eine Testabdeckung von 100%. Auch der neue Endpunkt *getSignedDownloadImageUrlsForTextDocumentIdForReview* der Klasse *ReviewLegacyEndpoint*, dessen Testabdeckung in Abbildung 2 dargestellt ist, hat eine Abdeckung von 100%. Die beiden Endpunkte zum Laden einer signierten Download-URL für die Bilder von Textdokumenten rufen die Methode *getSignedDownloadImageUrls* der Klasse *DocumentController* auf. Der Endpunkt *getSignedImageUploadUrlForTextDocumentId* zum Laden einer signierten Upload-URL für ein Bild in einem Textdokument ruft die Methode *getSignedImageUploadGCSUrl* der Klasse *DocumentController* auf. Diese beiden Methoden haben jeweils eine Codezeilentestabdeckung von 100%, was in Abbildung 3 zu sehen ist. Abbildung 4 zeigt, dass die von den beiden Methoden des *DocumentControllers* aufgerufenen Methoden *getSignedDownloadImageUrls* und *getSignedImageUploadUri* der Klasse *GcsUriSigner* auch eine Testabdeckung von 100% haben. Damit ist gezeigt, dass alle neu implementierten Endpunkte eine Codezeilentestabdeckung von 100% haben.

DocumentLegacyEndpoint

Element	Missed Instructions	Cov.
createDocumentFromTemplate(DocumentLegacyEndpoint.CreateFromTemplateRequest, AuthenticatedUser)		0 %
DocumentLegacyEndpoint()		100 %
getDocument(Long, AuthenticatedUser)		100 %
getDocuments(Long, ProjectType, AuthenticatedUser)		100 %
getSignedDownloadImageUrlsForTextDocumentId(Long, AuthenticatedUser)		100 %
getSignedDownloadUrlsForDocumentId(Long, AuthenticatedUser)		100 %
getSignedImageUploadUrlForTextDocumentId(Long, Long, AuthenticatedUser)		100 %
getSignedUploadUrlsForDocumentId(Long, AuthenticatedUser)		100 %

Abbildung 1: Codezeilentestabdeckung der Klasse *DocumentLegacyEndpoint*

ReviewLegacyEndpoint

Element	Missed Instructions	Cov.
getDocumentsForReview(Long, Long, Long, AuthenticatedUser)		100 %
getResultToReview(Long, Long, Long, ExerciseType, AuthenticatedUser)		100 %
getResultWithNewInstructorReview(Long, Long, ExerciseType, AuthenticatedUser)		100 %
getReviewFeedbackById(Long, Long, Long, AuthenticatedUser)		100 %
getReviewFeedbackByIdIfFinished(Long, Long, Long, AuthenticatedUser)		100 %
getReviewFeedbackByIdIfFinishedAndInstructor(Long, Long, Long, AuthenticatedUser)		100 %
getSignedDownloadImageUrlsForTextDocumentIdForReview(Long, Long, Long, Long, AuthenticatedUser)		100 %

Abbildung 2: Codezeilentestabdeckung der Klasse *ReviewLegacyEndpoint*

DocumentController

Element	Missed Instructions	Cov.
with(Context)		100 %
updateDocuments(List)		100 %
updateDocument(BaseDocument)		100 %
setDocumentsFromDifferentProjectsGroupedByName(List, ProjectType, Map, AuthenticatedUser)		100 %
saveAll(Collection)		100 %
save(BaseDocument)		100 %
removeDocuments(ProjectType, Long)		100 %
removeDocuments(BaseProject)		60 %
removeDocument(BaseDocument)		100 %
putDocumentToMemcache(BaseDocument)		100 %
listDocumentsForProject(ProjectType, Long)		100 %
insertDocument(BaseDocument)		100 %
getSignedImageUploadGCSUrl(BaseDocument, Long)		100 %
getSignedGCSUrls(BaseDocument, GcsUriSigner.UrlType)		100 %
getSignedDownloadImageUrls(TextDocument)		100 %

Abbildung 3: Codezeilentestabdeckung der Klasse *DocumentController*

GcsUriSigner

Element	Missed Instructions	Cov.
getSignedUris(HasGcsBlobs, GcsUriSigner.UrlType)		100 %
getSignedImageUploadUri(BaseDocument, Long)		100 %
getSignedDownloadImageUrls(TextDocument)		100 %

Abbildung 4: Codezeilentestabdeckung der Klasse *GcsUriSigner*

Literaturverzeichnis

Bühler, P., Schlaich, P., & Sinner, D. (2023). *HTML und CSS: Semantik - Design - Responsive Layouts* (2. Aufl.). Springer Vieweg.

Flick, U. (2008). *Triangulation - Eine Einführung* (2. Aufl.). VS Verlag für Sozialwissenschaften Wiesbaden.

Hartmann, N., & Zeigermann, O. (2020). *React: Grundlagen, fortgeschrittene Techniken und Praxistipps - mit TypeScript und Redux* (2. Aufl.). dpunkt.verlag.

Jaeger, A., Till und Metzger. (2011). *Open Source Software: Rechtliche Rahmenbedingungen der Freien Software* (3. Aufl.). C.H.Beck.

Kaufmann, A., Riehle, D., Krause, J., & Harutyunyan, N. (2023). A Solution for Automated Grading of QDA Homework. *Proceedings of the 56th Hawaii International Conference on System Sciences*, 44–53.

Kuckartz, U. (2014). *Mixed Methods - Methodologie, Forschungsdesigns und Analyseverfahren* (1. Aufl.). Springer VS Wiesbaden.

Post, U. (2021). *Besser coden: Best Practices für Clean Code* (2. Aufl.). Rheinwerk Verlag.

SOPHISTen. (2024). *Schablonen für alle Fälle* (6. Aufl.). SOPHIST GmbH.

Springer, S. (2020). *React: Das umfassende Handbuch* (1. Aufl.). Reihnwerk Verlag.