

# Verbesserung der kollaborativen Forschung im Web mit Yjs

MASTERARBEIT

Andreas Michael Hellmich

Eingereicht am 2. November 2023



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

Betreuer:

Julian Lehrhuber, M.Sc.  
Prof. Dr. Dirk Riehle, M.B.A.



Friedrich-Alexander-Universität  
Technische Fakultät



# 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, 2. November 2023

## Lizenz

Diese Arbeit unterliegt der Creative Commons Attribution 4.0 International Lizenz (CC BY 4.0), <https://creativecommons.org/licenses/by/4.0/>

---

Erlangen, 2. November 2023



# Abstract

This thesis describes an extension to **QDAcity**, a web application designed to support collaborative qualitative data analysis (QDA) for research teams. Especially for large data sets, the direct collaboration of researchers can significantly accelerate the analysis process. To better support the process, an adaptation of the Real-Time Collaboration Service (**RTCS**) which the past thesis by Dürsch, 2023 has started on will be continued. Thereby, as the component's name suggests, real-time synchronization between clients is enabled by means of the JavaScript library **Yjs**.

In this thesis, the adaptation of all relevant frontend components of **QDAcity** is described, starting from the previously implemented support for displaying and editing text documents. The first step was to introduce support for synchronization the code system, which is mandatory for qualitative data analysis. Furthermore, it will be described how previously existing data is migrated to new data structures that are optimized for use within **Yjs**. Since **QDAcity** is already used by various users, it is particularly important that no existing data is lost during this migration.

In addition to mere support for QDA, **QDAcity** also offers other features, such as recommendations for changes to the Codesystem (**CS**), which have been extended to include real-time synchronization.



# Zusammenfassung

Diese Arbeit beschreibt eine Erweiterung von **QDAcity**, einer Webanwendung zur Unterstützung der kollaborativen qualitativen Datenanalyse (QDA) für Forscherteams. Insbesondere bei großen Datenmengen kann die direkte Zusammenarbeit von Forschenden den Analysevorgang deutlich beschleunigen. Zur besseren Unterstützung des Prozesses wird eine bereits begonnene Anpassung des Echtzeit-Kollaborationsservice (engl. *Real-Time Collaboration Service*) (RTCS) fortgesetzt. Dabei wird, dem Komponentennamen entsprechend, mittels der JavaScript-Bibliothek **Yjs** Echtzeit-Synchronisation zwischen den Clients ermöglicht.

In dieser Arbeit wird die Anpassung aller relevanter Frontendkomponenten von **QDAcity** beschrieben, welche auf der zuvor implementierten Unterstützung für die Anzeige und Bearbeitung von Textdokumenten aufsetzt. Dabei wurde zunächst die Unterstützung für die Synchronisation des Codesystems (CSs) eingeführt, welches für die qualitative Datenanalyse (QDA) zwingend notwendig ist. Außerdem wird dargelegt, wie bisher vorhandene Daten auf neue, für die Verwendung in **Yjs** optimierte, Datenstrukturen migriert werden. Da **QDAcity** bereits von verschiedenen Nutzern verwendet wird, ist es bei dieser Migration besonders wichtig, dass keine bestehenden Daten verloren gehen.

Neben der reinen Unterstützung für die QDA bietet **QDAcity** zudem weitere unterstützende Features, wie Empfehlungen für Änderungen am CS, die im Rahmen der Anpassungen um Echtzeit-Synchronisation erweitert wurden.





# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Zielsetzung . . . . .	2
1.3	Gliederung . . . . .	2
<b>2</b>	<b>Hintergrund</b>	<b>3</b>
2.1	Qualitative Datenanalyse . . . . .	3
2.2	QDAcity . . . . .	3
<b>3</b>	<b>Anforderungen</b>	<b>9</b>
3.1	Funktionale Anforderungen . . . . .	9
3.2	Nicht-funktionale Anforderungen . . . . .	11
<b>4</b>	<b>Architektur</b>	<b>13</b>
4.1	Architektur vor kollaborativer Bearbeitung . . . . .	13
4.1.1	Hauptkomponenten . . . . .	13
4.1.2	Datenstruktur . . . . .	15
4.2	Aktuelle Architektur . . . . .	19
4.2.1	Hauptkomponenten . . . . .	19
4.2.2	Datenstruktur . . . . .	19
4.2.3	Datenmigration . . . . .	21
4.3	Geplante Architektur . . . . .	21
4.3.1	Hauptkomponenten . . . . .	21
4.3.2	Datenstruktur . . . . .	22
4.3.3	Datenmigration . . . . .	25
4.4	Notwendigkeit einer neuen Schnittstelle . . . . .	26
4.5	Mechanismen für sichere Server-Server-Kommunikation . . . . .	27
4.5.1	JSON Web Token . . . . .	27
4.5.2	mTLS . . . . .	28

<b>5</b>	<b>Design und Implementierung</b>	<b>31</b>
5.1	Implementierung kollaborativer Features mit Yjs . . . . .	31
5.1.1	Verwendung kollaborativer Daten im Frontend . . . . .	31
5.1.2	Anbindung weiterer Features an den CES . . . . .	38
5.2	Migration aller kollaborativer Daten . . . . .	39
5.2.1	Migration der Kodierungen . . . . .	39
5.3	Anpassungen des CES . . . . .	39
5.3.1	Prüfung der Autorisierung eines Nutzers . . . . .	40
5.3.2	Einführung einer Systemidentität . . . . .	41
<b>6</b>	<b>Evaluation</b>	<b>43</b>
6.1	Funktionale Anforderungen . . . . .	43
6.2	Nicht-funktionale Anforderungen . . . . .	45
<b>7</b>	<b>Hindernisse bei der Umsetzung</b>	<b>47</b>
<b>8</b>	<b>Fazit</b>	<b>49</b>
<b>Appendices</b>		<b>51</b>
A	Datenstruktur . . . . .	53
<b>Literaturverzeichnis</b>		<b>57</b>

# Abbildungsverzeichnis

2.1	QDAcity Projekt-Dashboard . . . . .	5
2.2	Modal zur Erstellung einer Aufgabe . . . . .	7
2.3	QDAcity Coding-Editor . . . . .	7
3.1	FunktionsMASTER mit Bedingung . . . . .	9
3.2	FunktionsMASTER ohne Bedingung . . . . .	10
3.3	BedingungsMASTER . . . . .	10
3.4	EigenschaftsMASTER . . . . .	11
3.5	UmgebungsMASTER . . . . .	12
4.1	Ursprüngliche QDAcity-Architektur . . . . .	14
4.2	Ursprüngliche Bucket-Struktur eines QDAcity-Buckets . . . . .	15
4.3	Aktuelle QDAcity-Architektur . . . . .	20
4.4	Aktuelle Bucket-Struktur eines QDAcity-Buckets . . . . .	20
4.5	Geplante QDAcity-Architektur . . . . .	22
4.6	Geplante Bucket-Struktur eines QDAcity-Buckets . . . . .	23
4.7	Normaler Ablauf TLS-Handshake . . . . .	29
4.8	Erweiterter Ablauf mTLS-Handshake . . . . .	29
4.9	Open Systems Interconnection (OSI)-Schichtenmodell mit Protokollen . . . . .	30
5.1	Anzeige aktiver Projekt-Nutzer in der Editor-Toolbar . . . . .	35
5.2	Anzeige aktiver Nutzer in der Dokumentübersicht . . . . .	35
5.3	Anzeige einer Selektion im Kodierungseditor (ohne Caret) . . . . .	37
5.4	Anzeige einer Selektion im Texteditor (mit Caret) . . . . .	37
5.5	Anzeige der Position eines Nutzers im Texteditor . . . . .	37

1	Vollständige Bucket-Struktur eines QDAcity-Buckets vor Einführung des kollaborativen Editierens . . . . .	53
2	Vollständige Bucket-Struktur eines QDAcity-Buckets nach Einführung des kollaborativen Editierens . . . . .	54
3	Geplante vollständige Bucket-Struktur eines QDAcity-Buckets . .	55

# Listings

4.1	Repräsentation der <code>content</code> HTML-Struktur . . . . .	15
4.2	Repräsentation der ursprünglichen <code>Codedaten</code> -Struktur als anno- tiertes JSON . . . . .	16
4.3	Repräsentation der ursprünglichen <code>Kodierungsdaten</code> -Struktur für Textkodierungen als annotiertes JSON . . . . .	17
4.4	Repräsentation der ursprünglichen <code>Kodierungsdaten</code> -Struktur für PDF-Textkodierungen als annotiertes JSON . . . . .	18
4.5	Repräsentation der ursprünglichen <code>Kodierungsdaten</code> -Struktur für PDF-Bereichskodierungen als annotiertes JSON . . . . .	18
4.6	Repräsentation der geplanten <code>Kodierungsdaten</code> -Struktur als an- notiertes JSON . . . . .	24
4.7	Repräsentation der geplanten <code>Kodierungsdaten</code> -Struktur für PDF- Bereichskodierungen als annotiertes JSON . . . . .	24
4.8	Repräsentation der geplanten <code>Kodierungsdaten</code> -Struktur für PDF- Textkodierungen als annotiertes JSON . . . . .	25
5.1	Handling von Änderungen . . . . .	33
5.2	Preview Update . . . . .	34
5.3	Repräsentation der <code>Awareness-Daten</code> -Struktur für Nutzer im Projekt	36
5.4	Repräsentation der <code>Awareness-Daten</code> -Struktur für den Editorsta- tus eines Nutzers im Textdokument . . . . .	37



# Akronyme

<b>CA</b>	Certificate Authority
<b>CES</b>	Service zur Kollaborativen Bearbeitung (engl. <i>Collaborative Editing Service</i> )
<b>CS</b>	Codesystem
<b>FA</b>	funktionelle Anforderungen
<b>GCP</b>	Google Cloud Platform
<b>GCS</b>	Google Cloud Storage
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IaaS</b>	Infrastructure as a Service
<b>ICA</b>	Übereinstimmung zwischen Kodierern (engl. <i>Intercoder Agreement</i> )
<b>JSON</b>	JavaScript Object Notation
<b>JWT</b>	JavaScript Object Notation Web Token
<b>mTLS</b>	mutual Transport Layer Security
<b>NFA</b>	nichtfunktionelle Anforderungen
<b>NLP</b>	Verarbeitung natürlicher Sprache (engl. <i>Natural Language Processing</i> )
<b>OAuth</b>	Open Authorization
<b>OSI</b>	Open Systems Interconnection
<b>PaaS</b>	Platform as a Service
<b>QDA</b>	qualitative Datenanalyse
<b>RBAC</b>	rollenbasierte Zugriffskontrolle (engl. <i>Role-Based Access Control</i> )

<b>REST-API</b>	Representational State Transfer Application Programming Interface
<b>RFC</b>	Request for Comments
<b>RTCS</b>	Echtzeit-Kollaborationsservice (engl. <i>Real-Time Collaboration Service</i> )
<b>SaaS</b>	Software as a Service
<b>SPA</b>	Single-Page Application
<b>TLS</b>	Transport Layer Security



# 1 Einleitung

Qualitative Datenanalyse (QDA) ist ein wesentlicher Bestandteil aller qualitativer Studien zur Auswertung von Freitext-Antworten in Umfragen, Interviews oder anderen unstrukturierten Daten. Eine weit verbreitete qualitative Analyse-methode stellt hierbei die kategorienbildende Analyse dar (Springer, n. d.). Bei dieser werden zunächst Codes definiert, welche dann als Kodierung einzelnen Datensegmenten zugeordnet werden. Im Laufe der Analyse können weitere Codes hinzugefügt und ein hierarchischer Beziehungsbaum herausgearbeitet werden. Um Forschende bei dieser Arbeit zu unterstützen, bietet **QDAcity** als cloudbasierte Browseranwendung ein breites Toolset zur Unterstützung der kategorienbildenden Analyse.

## 1.1 Motivation

Gerade bei großen Forschungsprojekten mit mehreren beteiligten Forschern fehlen in **QDAcity** jedoch einige wichtige Features. So ist es nicht möglich, Änderungen von anderen Mitarbeitern in Echtzeit zu synchronisieren. Woran genau ein anderer Nutzer arbeitet, wird ebenfalls nicht angezeigt.

Insbesondere im Vergleich mit anderen Online-Editoren mit Kollaborationsfunktion wie Office 365 oder Google Docs können diese fehlenden Features zu einer Ablehnung durch die Nutzer führen. Durch den hohen Marktanteil der zuvor genannten Tools ist anzunehmen, dass die Live-Synchronisation in Online-Editoren für die meisten Anwender bereits als notwendige Basisfunktionalität angesehen wird. Darüber hinaus wird durch die Echtzeitsynchronisation die Effizienz der QDA deutlich erhöht, was wiederum zu einer besseren Nutzergewinnung führen kann.

Um dieser Erwartung gerecht zu werden, soll **QDAcity** im Rahmen dieser Arbeit so erweitert werden, dass eine Echtzeit-Synchronisation aller relevanter Daten erfolgt.

### 1.2 Zielsetzung

Im Rahmen dieser Arbeit soll die von Dürsch (2023) begonnene Einführung von Echtzeit-Synchronisation innerhalb von `QDAcity` weitergeführt werden. Hierfür soll der Service zur Kollaborativen Bearbeitung (engl. *Collaborative Editing Service*) (`CES`) weiterentwickelt werden, sodass der aktuelle Echtzeit-Kollaborationsservice (engl. *Real-Time Collaboration Service*) (`RTCS`) durch ihn ersetzt werden kann. Zudem soll das Frontend vollständig an die Verwendung des `CES` angepasst werden.

Außerdem soll eine Erweiterung für den `CES` konzipiert und bei ausreichend verbleibender Zeit implementiert werden, sodass das Java-Backend von `QDAcity` direkt auf den bereits eingeführten kollaborativen Daten im `Yjs`-Format arbeiten kann. Dafür soll eine Representational State Transfer Application Programming Interface (REST-API) entworfen werden, die dem Backend alle notwendigen Informationen liefert.

Im Zuge der zuvor beschriebenen Änderungen sollen alle kollaborativen Daten zudem in einem einheitlichen Schema abgelegt werden.

### 1.3 Gliederung

Diese Arbeit gliedert sich neben der Einleitung in 6 weitere Kapitel:

In Kapitel 2 *Hintergrund* werden grundlegende Softwarekomponenten, Vorgehensweisen und Standards vorgestellt, die für diese Arbeit eine zentrale Rolle spielen. Dabei wird auch `QDAcity` und dessen Funktionsumfang ausführlich beschrieben. Darauf folgend werden in Kapitel 3 *Anforderungen* funktionelle Anforderungen (FA) und nichtfunktionelle Anforderungen (NFA) dargelegt, welche an die für `QDAcity` zu entwickelnde Erweiterung gestellt werden. In Kapitel 4 *Architektur* wird hiernach die aktuelle Architektur von `QDAcity` aufgezeigt, bevor die vorgesehenen Änderungen erläutert werden. Daraufhin wird in Kapitel 5 *Design und Implementierung* Einblick in die Designentscheidungen während der Erweiterung gegeben und einige Aspekte der Implementierung näher erläutert. Die Umsetzung der in *Anforderungen* aufgelisteten Anforderungen wird in Kapitel 6 *Evaluation* bewertet. Im vorletzten Kapitel 7 *Hindernisse bei der Umsetzung* wird kurz auf Probleme eingegangen, die verzögernde Auswirkungen auf die Umsetzung hatten. Abschließend werden die Ergebnisse der Arbeit in Kapitel 8 *Fazit* nochmal zusammengefasst, sowie eine Übersicht über die während der Implementierung aufgetretenen Schwierigkeiten und ein Ausblick auf mögliche Folgearbeiten gegeben.

## 2 Hintergrund

### 2.1 Qualitative Datenanalyse

QDA beschreibt den Prozess des Beschreibens, Klassifizierens und Verknüpfens der beobachteten Phänomene mit den Konzepten der zugrundeliegenden Forschung. Um auf diese Weise das untersuchte Phänomen beschreiben zu können, muss zunächst ein konzeptuelles Framework erstellt und die vorliegenden Daten klassifiziert werden. Danach können die einzelnen Konzepte miteinander in Verbindung gebracht werden (Graue, 2015).

Dabei umfassen qualitative Daten verschiedenste Datenquellen, in welchen das Augenmerk auf der Bedeutung und nicht auf reinen Zahlen liegt. Dies umfasst unter anderem Beobachtungsprotokolle, Interviews, Videos, Texte oder Notizen (Graue, 2015). Diese Arbeit bezieht sich dabei auf die Unterstützung von Forschenden bei der Auswertung von qualitativen Daten, nicht bei deren Generierung.

Um mit qualitativen Daten arbeiten zu können, müssen diese, anders als bei quantitativen Daten, zunächst analysiert und strukturiert werden. Dafür eignet sich die Kategorisierung oder Kodierung der Daten (Graue, 2015), bei der einzelnen (Text-)Segmente eine Kategorie zugeordnet wird, um diese später gezielt auswerten zu können.

### 2.2 QDAcity

QDAcity ist eine von der Professur für Open Source Software der Friedrich-Alexander-Universität Erlangen-Nürnberg entwickelte browserbasierte Software as a Service (SaaS)-Anwendung für QDA. Sie stellt eine Umgebung für kollaborative QDA für wissenschaftliche Teams dar, in der mehrere Nutzer in Echtzeit kollaborativ an einer Datei arbeiten können. Dabei kann einer Textpassage eine Kategorie (folgend **Code**) zugeordnet werden. Eine solche Zuordnung wird als **Kodierung** bezeichnet.

## 2. Hintergrund

---

QDAcity ist dabei grundsätzlich in 3 Hauptkomponenten zu untergliedern:

- **Frontend:** Eine auf React basierende Single-Page Application (SPA), die überwiegend in JavaScript entwickelt ist und jegliche für die Nutzerinteraktion notwendige Logik beinhaltet.
- **Backend:** Ein auf Java und Google App Engine<sup>1</sup> basierender Server, der die Datenhaltung und Nutzerauthentifikation umfasst.
- **RTCS:** Ein in Javascript geschriebener Server, der verschiedene Awareness-Funktionen (z.B. aktive Nutzer im Projekt) bereitstellt, die kollaborativen Sessions verwaltet und regelmäßig darin vorgenommene Änderungen persistiert.

Zur Umsetzung dieser Komponenten setzt QDAcity stark auf die Infrastructure as a Service (IaaS)- und Platform as a Service (PaaS)-Angebote der Google Cloud Platform (GCP). So werden Google App Engine und Google Cloud Run<sup>2</sup> zum Betrieb der Softwarekomponenten verwendet. Um die Daten zu speichern wird außerdem auf Google Cloud Datastore<sup>3</sup> und Google Cloud Storage<sup>4</sup> gesetzt. Für weitere Funktionalität werden außerdem noch Google BigQuery<sup>5</sup>, Google Cloud Endpoints<sup>6</sup>, Google Speech-to-Text<sup>7</sup> und Redis<sup>8</sup> verwendet.

Dabei bietet QDAcity den Nutzern eine Vielzahl an Funktionen:

- Anmeldung über E-Mail und Passwort oder Google Open Authorization (OAuth)
- Organisation aller Daten in "Projekten" zur einfacheren Verwaltung
- Projektteilnehmer mit Rechtevergabe (Owner, Organizer, Editor, Viewer)
- Projektrevisionen und Revisionsverlauf
- Projektdashboard mit statistischen Informationen
- Hierarchisches Codesystem
- Möglichkeiten zur Analyse, Anpassung und Erweiterung der Codes mittels zusätzlicher Daten oder Metadaten
- Kodierungseditor für Text und PDF-Dateien
- Speech-To-Text Feature zur Transkription von Interviews

---

<sup>1</sup> <https://cloud.google.com/appengine>

<sup>2</sup> <https://cloud.google.com/run>

<sup>3</sup> <https://cloud.google.com/datastore>

<sup>4</sup> <https://cloud.google.com/storage>

<sup>5</sup> <https://cloud.google.com/bigquery>

<sup>6</sup> <https://cloud.google.com/endpoints>

<sup>7</sup> <https://cloud.google.com/speech-to-text>

<sup>8</sup> <https://redis.com/cloud-partners/google/>

- To-Do Listen zur einfacheren Organisation von Aufgaben
- UML-Editor zur Darstellung von Code-Beziehungen

Um die Zusammenhänge der einzelnen Komponenten und die damit einhergehenden Einschränkungen in dieser Arbeit besser verstehen zu können, ist es wichtig, dass man sich mit der Nutzung von QDAcity zur Kodierung eines Projekts vertraut macht. Daher wird im Folgenden kurz ein übliches Anwendungsszenario vorgestellt.

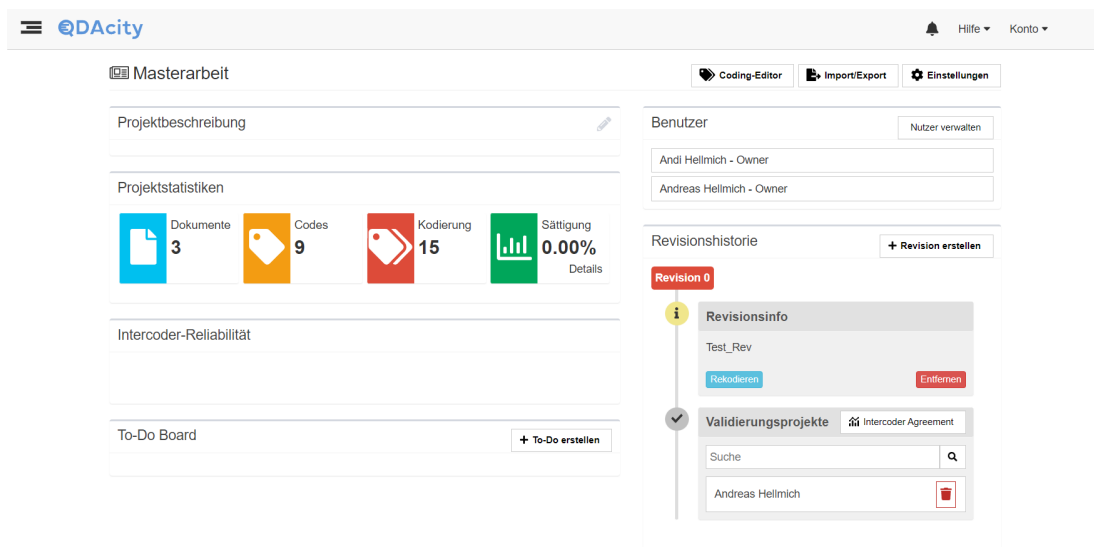


Abbildung 2.1: QDAcity Projekt-Dashboard

Zu Beginn muss ein neues Projekt angelegt werden. Danach befindet man sich im Projekt-Dashboard *Abbildung 2.1*. Diese Ansicht ist in zwei Spalten gegliedert. In der ersten Spalte findet sich der Großteil der allgemeinen Informationen über das Projekt:

- Der Projektname
- Eine Projektbeschreibung (so vorhanden)
- Projektstatistiken: Anzahl von Dokumenten, Codes, Kodierungen und die Sättigung
- Eine Übersicht über die neuesten Protokolle zur Übereinstimmung zwischen Kodierern (engl. *Intercoder Agreement*) (ICA)
- Sowie ein projektweites To-Do Board

In der rechten Spalte finden sich zunächst Schaltflächen für den Wechsel in den Coding Editor, zum Import oder Export von Daten und zu den Projekteinstel-

lungen. Darunter werden alle Nutzer mit Zugriff auf das Projekt angezeigt. Bei ausreichenden Rechten findet sich hier auch eine Schaltfläche zum Wechsel in die Nutzerverwaltung. Darunter findet sich die Möglichkeit, Revisionen anzulegen, aus Revisionen Validierungsprojekte zu erzeugen ("Rekodieren") und in die ICA-Ansicht zu wechseln.

QDAcity unterscheidet dabei grundsätzlich zwischen vier Projektarten:

- **Projekt (PROJECT):** das eigentliche Projekt, in dem die aktuellen Dokumente, sowie die aktuellen Codes und Kodierungen gespeichert sind.
- **Projektrevision (REVISION):** eine nicht veränderbare Kopie eines Projekts. Diese spiegelt jeweils den Stand eines Projekts zum Zeitpunkt des Erstellens der Revision wieder. Sie beinhaltet eine Kopie aller zum Erstellungszeitpunkt existierender Dokumente, Codes und Kodierungen.
- **Validierungsprojekt (VALIDATION):** Ein Validierungsprojekt kann aus einer Revision erstellt werden. Es beinhaltet Dokumente und Codes, aber nicht die einzelnen Kodierungen und dient dazu, dass die Dokumente ein weiteres Mal kodiert werden können. Auf Validierungsprojekten können ICA-Berichte erstellt werden, um die Übereinstimmung der Kodierungen zwischen der Revision aus welcher das Validierungsprojekt erstellt wurde und dem Validierungsprojekt selbst auszuwerten.
- **Übungen (EXERCISE):** Um im Vorlesungs- oder Übungsbetrieb QDA lehren und effizient korrigieren zu können, können Kurse angelegt werden. Für diese kann eine Projektrevision in eine Aufgabe überführt werden, wobei zwischen zwei Aufgabentypen unterschieden wird:
  - **Codebuch-Nutzung:** Dabei werden die Dokumente und das Codesystem kopiert; Kodierungen werden nicht übernommen. Die Auswertung erfolgt durch einen automatisierten Vergleich der Kodierungen in der Revision und der Übung.
  - **Codebuch-Erstellung:** Hierbei werden nur die Dokumente übernommen. Das erstellte Codesystem wird mit dem der Ursprungsrevision verglichen.

Diese Aufgaben besitzen eine Deadline und können benannt und für die Teilnehmer beschrieben werden (*Abbildung 2.2*).

Im Normalfall wird vom Projektdashboard in den Coding-Editor (*Abbildung 2.3*) des eigentlichen Projekts oder eines Validierungsprojekts gewechselt.

Hier können in der Header-Leiste die einzelnen Editoren gewechselt werden. In der linken Spalte gibt es einen Abschnitt zur Verwaltung der Dokumente, sowie einen zur Verwaltung und Anwendung der Kodierungen.



## 2. Hintergrund

---



# 3 Anforderungen

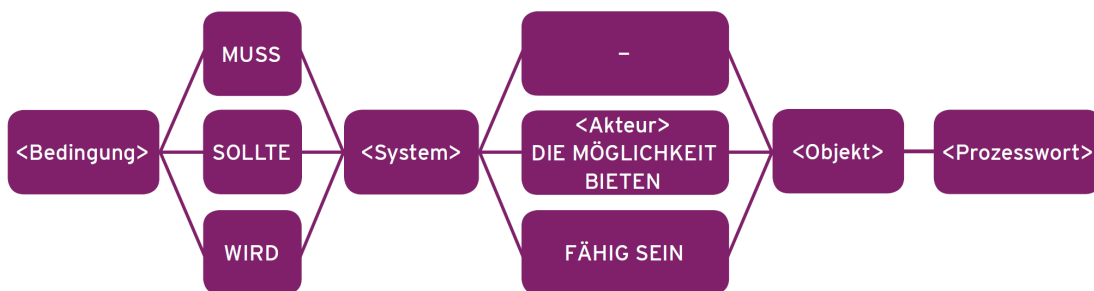
In diesem Kapitel werden alle Anforderungen beschrieben, welche an die geplante Erweiterung von QDAcity gestellt werden.

Hierbei wird zunächst jeweils das zur Beschreibung verwendete Schema beschrieben, bevor die einzelnen Anforderungen aufgeführt sind. Es werden zuerst die FA aufgezählt, bevor auf die NFA eingegangen wird.

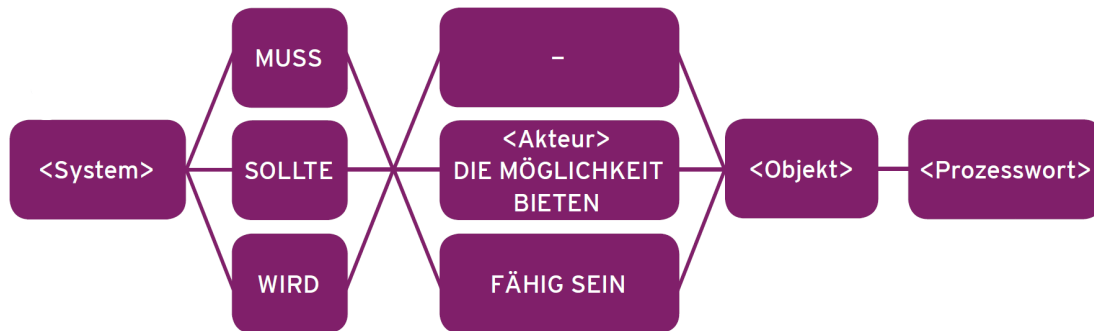
## 3.1 Funktionale Anforderungen

Die FA sind nach dem FunktionsMASTER von Rupp und Joppich (2014) formuliert. Das verwendete Schema ist in *Abbildung 3.1* und *Abbildung 3.2* dargestellt. Dabei stellt *Abbildung 3.1* eine Variation von *Abbildung 3.2* dar, in der zusätzlich eine Bedingung eingeführt ist, die definiert, wann die beschriebene Anforderung gelten soll.

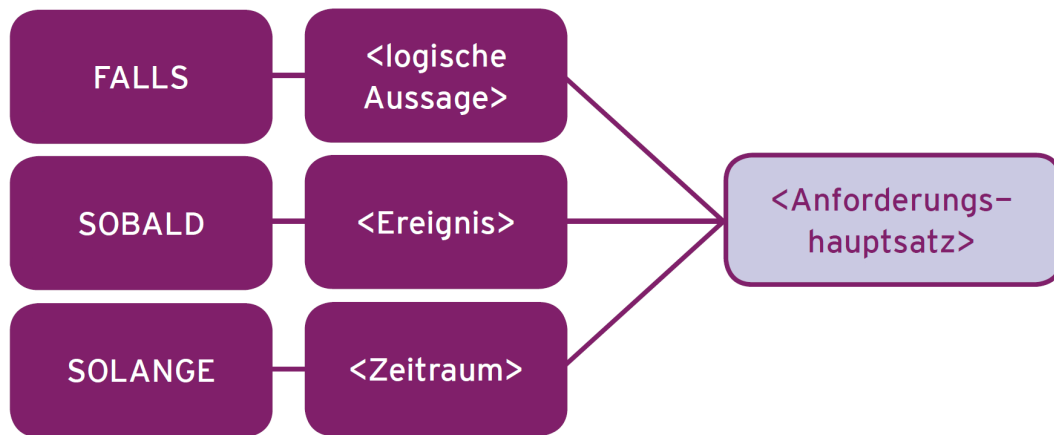
Das Schema für diese Bedingungen ist in *Abbildung 3.3* definiert.



**Abbildung 3.1:** FunktionsMASTER mit Bedingung (Rupp und Joppich, 2014, S. 224)



**Abbildung 3.2:** FunktionsMASTER ohne Bedingung (Rupp und Joppich, 2014, S. 220)



**Abbildung 3.3:** BedingungsMASTER (Rupp und Joppich, 2014, S. 241)

**FA-1:** Yjs sollte alle Daten, die für die Nutzer von Relevanz sind, in Echtzeit zwischen den Nutzern synchronisieren.

**FA-1.1:** Sobald Änderungen am Codesystem wie Erstellen, Umbenennen, Änderungen an der hierarchischen Struktur oder Löschen von Codes vorgenommen werden, muss Yjs diese Änderungen synchronisieren.

**FA-1.2:** Sobald Änderungen an den mit dem Codesystem verknüpften Entitäten Memo, CodeRelation und CodebookEntry vorgenommen werden, muss Yjs diese Änderungen synchronisieren.

**FA-1.3:** Sobald Änderungen am Dokumentinhalt wie Kodierungen oder Textanpassungen bei Textdokumenten vorgenommen werden, muss Yjs diese synchronisieren.

**FA-1.4:** Sobald Änderungen an Dokument-Metadaten, also das Erstellen, Umbenennen oder Löschen, vorgenommen werden, sollte Yjs andere Clients über diese Änderungen informieren.

**FA-2:** Yjs sollte Awareness-Informationen über den Nutzer an andere Clients senden.

**FA-2.1:** Falls ein Nutzer im Projekt aktiv ist, muss diese Information in der Nutzeranzeige angezeigt werden.

**FA-2.2:** Sobald ein Nutzer ein Dokument geöffnet hat, sollte dies in der Dokumentenanzeige angezeigt werden.

**FA-2.3:** Sobald der Texteditor geöffnet ist, muss der Editor die Textselektierung eines Nutzers anzeigen.

**FA-2.4:** Sobald ein Nutzer im Texteditierungsmodus ist, muss der Editor den Cursor des anderen Nutzers anzeigen.

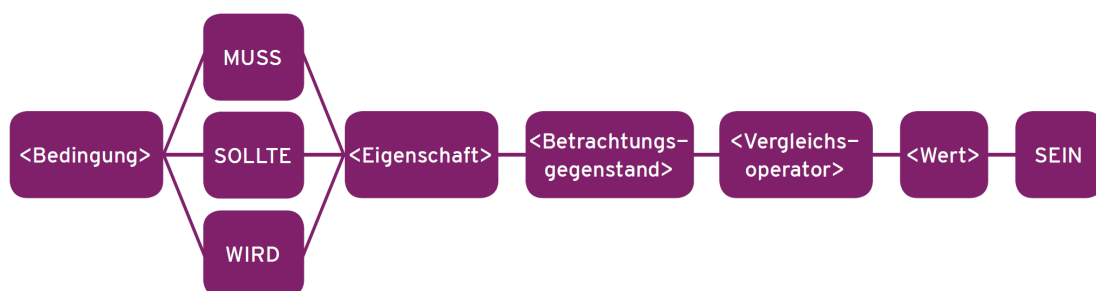
**FA-2.5:** Sobald ein Bereich eines PDFs angewählt ist, sollte der Editor die Selektion des anderen Nutzers anzeigen.

**FA-3:** Der CES muss fähig sein, nur berechtigten Servern von QDAcity die angeforderten Daten zu liefern und nicht autorisierte Anfragen abzulehnen.

**FA-3.1:** Zur Authentifizierung einer Anfrage sollen keine Autorisierungs-Rundläufe notwendig sein.

## 3.2 Nicht-funktionale Anforderungen

Die NFA sind ebenfalls nach der Vorlage von Rupp und Joppich (2014) formuliert. Dabei wird das EigenschaftsMASTER wie in *Abbildung 3.4* verwendet, wenn NFA Eigenschaften für die Erweiterungen des Backends oder des CES beschreiben.



**Abbildung 3.4:** EigenschaftsMASTER (Rupp und Joppich, 2014, S. 235)

Werden NFA beschrieben, die von der Operationsumgebung des CES erfordert werden, so werden diese anhand des UmgebungsMASTERs aus *Abbildung 3.5* formuliert.

### 3. Anforderungen

---

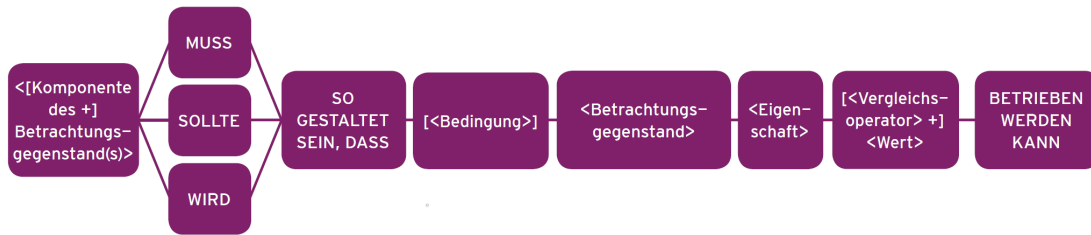


Abbildung 3.5: UmgebungsMASTER (Rupp und Joppich, 2014, S. 237)

**NFA-1:** Der CES muss so gestaltet sein, dass er innerhalb der von Google Cloud Run zur Verfügung gestellten Umgebung ausgespielt und betrieben werden kann.

**NFA-1.1:** Google Cloud Run bietet ein Maximum von 32 Gigabyte Arbeitsspeicher pro Instanz.<sup>1</sup>

**NFA-1.2:** Websocket-Verbindungen unter Google Cloud Run müssen innerhalb einer Stunde abgeschlossen sein.<sup>1</sup>

**NFA-2:** Die GitLab Deployment-Pipeline muss so gestaltet sein, dass der CES bei jedem Deployment von QDAcity mit ausgespielt wird.

**NFA-2.1:** Die GitLab Deployment-Pipeline muss so gestaltet sein, dass die Environment-Variablen des CES auf die aktuelle Backend-URL verweisen.

**NFA-3:** Der CES muss so gestaltet sein, dass alle bestehenden Features erhalten bleiben und in Zukunft leicht neue Features implementiert werden können.

**NFA-3.1:** Der CES muss so gestaltet sein, dass jegliche bestehende Funktionalität erhalten bleibt.

**NFA-3.2:** Der CES muss so gestaltet sein, dass zukünftig weitere Dokumenttypen synchronisiert werden können.

**NFA-3.3:** Der CES muss so gestaltet sein, dass er in Zukunft Offline-Bearbeitung unterstützen kann.

**NFA-3.4:** Der CES muss so gestaltet sein, dass in Zukunft auch weitere Features leicht unterstützt werden können.

---

<sup>1</sup> <https://cloud.google.com/run/quotas>

# 4 Architektur

In diesem Kapitel wird die Architektur von `QDAcity` beschrieben. Da diese Arbeit eine Brücke zwischen zwei vorangegangenen Arbeiten bildet, müssen bei Betrachtung der Architektur drei Versionen berücksichtigt werden.

Daher wird zunächst in *4.1* auf die Architektur eingegangen, wie sie nach den Anpassungen von Anavatti (2022) vorlag. Folgend wird in *4.2* der Zustand nach Abschluss der Implementierung des kollaborativen Bearbeitungsmodus beschrieben. In *4.3* werden die geplanten Anpassungen und deren Auswirkungen auf die aktuelle Struktur skizziert. Abschließend wird in *4.4* beschrieben, weshalb der `CES` vor seiner Produktionseinführung um eine REST-API erweitert werden muss. Dafür werden verschiedene Authentifizierungsmechanismen beschrieben, bevor eine Empfehlung für die Verwendung innerhalb der Schnittstelle abgegeben wird.

## 4.1 Architektur vor kollaborativer Bearbeitung

### 4.1.1 Hauptkomponenten

In der ursprünglichen Architektur finden sich die in *2.2* beschriebenen drei Komponenten gut abgrenzbar wieder (*Abbildung 4.1*).

Im Frontend werden alle Daten in der React-SPA angezeigt, die Texteditor-Funktionalität wird über einen `SlateJS`<sup>1</sup>-Editor abgebildet. Dabei ist die Datenquelle für das Frontend zweigeteilt:

- Eine signierte URL zum Abruf des Dokuments vom Google Cloud Storage (GCS) und die zum Dokument gehörigen Metadaten ruft das Frontend via HTTP-Anfragen direkt vom Java-Backend-Server ab. Das Dokument selbst wird folgend direkt vom GCS geladen.
- Auch die Coding-Informationen bezieht das Frontend direkt vom Backend. Hier werden Änderungen allerdings über Websocket-Verbindungen an den RTCS übertragen. In diesem Fall werden alle Änderungen anschließend vom RTCS an das Backend gesendet und dort persistiert. Kann die Websocket-Verbindung nicht hergestellt werden, so werden Änderungen direkt an das

---

<sup>1</sup> <https://docs.slatejs.org/>

Backend gesendet. Im Normalfall werden alle vom Nutzer vorgenommenen Änderungen periodisch mit dem RTCS synchronisiert. Aufgrund der zwei Übertragungswege und der nur periodischen Synchronisation kann es hier dazu kommen, dass sich Änderungen gegenseitig überschreiben.

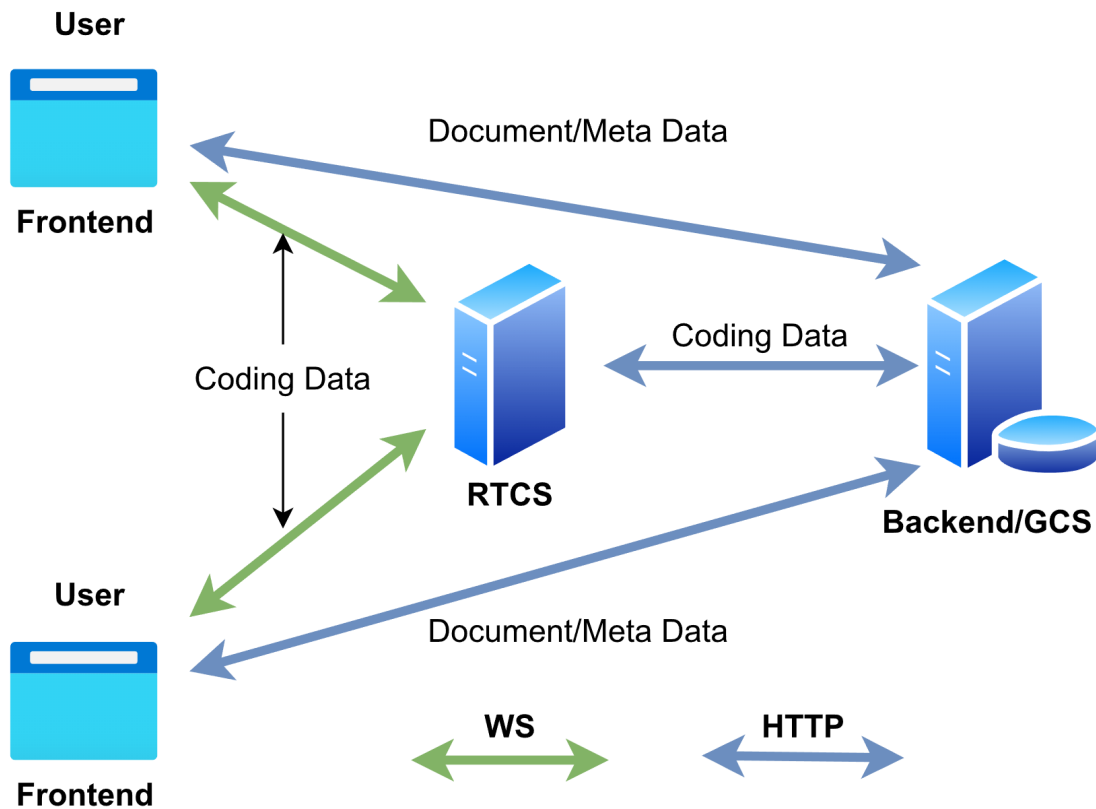


Abbildung 4.1: Ursprüngliche QDAcity-Architektur (Dürsch, 2023, S. 45)

Der RTCS ist ein NodeJS `express`<sup>2</sup> Server, der auf der Google Cloud Run<sup>3</sup>-Plattform gehostet wird. Er verwaltet die Codes und Kodierungsinformationen und liefert außerdem Informationen zu allen Nutzern, die aktuell im jeweiligen Projekt online sind.

Der Backend-Java-Server verwendet Google Cloud Endpoints<sup>4</sup>. Er liefert alle notwendigen Kodierungsdaten für aktuell geöffnete Projekte an das Frontend und den RTCS und dient der eigentlichen Datenverwaltung. Außerdem verwaltet das Backend den Zugriff auf die Dokumente in den Google Cloud Storage<sup>5</sup> Buckets und persistiert alle weiteren Daten im Google Cloud Datastore<sup>6</sup>.

<sup>2</sup> <https://expressjs.com/de/>

<sup>3</sup> <https://cloud.google.com/run>

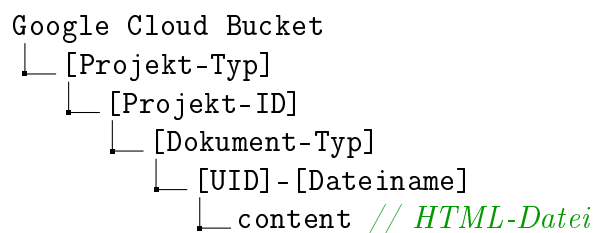
<sup>4</sup> <https://cloud.google.com/endpoints>

<sup>5</sup> <https://cloud.google.com/storage>

<sup>6</sup> <https://cloud.google.com/datastore>

### 4.1.2 Datenstruktur

Die Datenstruktur der Google Cloud Buckets entspricht dem Muster in *Abbildung 4.2*. Dabei beschreibt **Projekt-Typ** jeweils einen der vier in 2.2 beschriebenen Projekttypen (PROJECT, REVISION, VALIDATION, EXERCISE). Eine volle Repräsentation unter Berücksichtigung der einzelnen Projekttypen ist im Anhang in *Abbildung 1* zu finden. **Dokument-Typen** umfassen zum beschriebenen Zeitpunkt TEXT, PDF und TRANSCRIPTION. Die UID vor dem Dateinamen wird beim Erstellen einer Datei zufällig generiert, um bei gleich benannten Dokumenten Überschreibungen zu verhindern.



**Abbildung 4.2:** Ursprüngliche Bucket-Struktur eines QDAcity-Buckets

In der `content`-Datei befindet sich eine HTML-Repräsentation der Text-Datei. Diese ist zunächst in Paragraphen gegliedert. Jeder Paragraph enthält mindestens einen `span`, wobei jeder `span` mit einem innerhalb des Dokuments einmaligen Wert des `codingkey`-Attributs zur Identifizierung versehen ist. In *Listing 4.1* wird diese Struktur dargestellt. Darin ist außerdem zu erkennen, dass Text-Stile über reguläre HTML-Tags (`<b>`, `<u>`, `<i>`) definiert werden. Dafür wird der jeweilige `span` geteilt. Er wird als jeweils ein `span` vor und nach dem Stilelement überführt. Außerdem wird ein weiterer `span` innerhalb des Stilelements erstellt. Jeder Textabschnitt mit gleichbleibender Formatierung wird somit durch einen eigenen `span` (mit eigenem Wert im `codingkey`) repräsentiert.

```

<p>
  <i><span codingkey="3">Lorem ipsum dolor sit amet, consetetur
  ↪ sadipscing elit, </span></i>
  <b><span codingkey="2">sed diam nonumy eirmod tempor invidunt ut
  ↪ labore et dolore magna aliquyam erat, </span></b>
  <span codingkey="5">sed diam voluptua. </span>
  <u><span codingkey="4">At vero eos et accusam </span></u>
  <span codingkey="1">et justo duo dolores et ea rebum. </span>
</p>

```

**Listing 4.1:** Repräsentation der `content` HTML-Struktur<sup>7</sup>

<sup>7</sup> Zeilenumbrüche und Einrückung für bessere Lesbarkeit eingefügt.

Auf dem in dieser Struktur befindlichen Text können die Nutzer nun im Rahmen der QDA Kodierungen vornehmen. Dabei werden die verwendeten Codes und die dabei erstellten Kodierungen im Google Cloud Datastore gespeichert.

Die Struktur eines Codes wird von *Listing 4.2* repräsentiert. Hierbei beschreibt der Key, welcher als Kombination von ID/Name und dem Datentyp jeden Eintrag im Datastore eindeutig identifiziert, die Identität des Eintrags. Sie wird innerhalb von QDAcity zur Identifizierung von Datensätzen verwendet.

Neben dieser Id besitzt jeder Code außerdem eine `localId`, welche einmalig pro Codesystem (CS) vergeben wird und innerhalb von einem CS als Referenz dient. Diese nicht-globale Id wurde eingeführt, um eine einfachere CS-Duplikation zu ermöglichen. Dadurch müssen während des Kopiervorgangs nur neue Ids für die neuen Elemente erstellt werden. So können alle Referenzen von Kodierungen auf ihre Codes unmodifiziert übernommen werden, da die Codes ihre als Referenz verwendete lokale Id behalten.

---

```
{
  "key": string,
  "author": string,
  "codeBookEntry": Datastore Referenz Code/CodeBookEntry,
  "codesystemID": int,
  "color": HEX-Colorcode,
  "isDraft": boolean,
  "isFavored": boolean,
  "localId": int,
  "memo": string,
  "message": string,
  "mmElementIDs": [Referenz Code/MetaModel Entity],
  "parentID": int,
  "relationshipCode": Referenz Code/Code Relation
  "relations": [Referenz Code/Code Relation],
}
```

---

**Listing 4.2:** Repräsentation der ursprünglichen Codedaten-Struktur als annotiertes JSON

Die Kodierungen werden, analog zu den Codes, als eigene Entitäten im Datastore abgelegt. Auch hier ist das Identitäts-Feld ID/Name vorhanden. Eine Kodierung referenziert außerdem immer die Id seines zugehörigen CSs, des Dokuments in dem sie sich befindet und die `localId` des Codes, den sie repräsentiert.

Bei TEXT-Dokumenten (*Listing 4.3*) wird die eigentliche Position einer Kodierung innerhalb des Dokuments in vier Eigenschaften abgespeichert, wobei eine Position im Dokument durch je zwei Eigenschaften beschrieben wird:

Der Startpunkt wird durch `startKey` und `pos1` identifiziert. Der `startKey` referenziert dabei einen `codingKey` im Dokument, `pos1` den Offset in Zeichen nach Beginn des dadurch identifizierten `spans`. Analog wird der Endpunkt durch `endKey` und `pos2` definiert.



Darüber hinaus wird eine `preview` der kodierten Text-Passage gespeichert, um diese unabhängig vom zugehörigen Dokument darstellen zu können.

---

```

{
  "Key": string ,
  "ID/Name": int ,
  "DISCRIMINATOR": string , // Voll qualifizierender JAVA class name
  "author": string ,
  "codesystemID": int ,
  "documentId": int ,
  "endKey": int ,
  "localCodeId": int ,
  "pos1": int ,
  "pos2": int ,
  "preview": string ,
  "projectId": int ,
  "projectType": ProjectTypeEnum ,
  "startKey": int ,
  "timestamp": DateTime
}

```

---

**Listing 4.3:** Repräsentation der ursprünglichen Kodierungsdaten-Struktur für Textkodierungen als annotiertes JSON

Die Struktur der PDF-Dokument-Kodierungen (*Listing 4.4*) entspricht der der TEXT-Dokumente, bis auf die Eigenschaften zur Positionierung der Kodierung. Bei PDFs gibt es zwei verschiedene Kodierungstypen:

- **Textkodierungen:** Diese entsprechen den Kodierungen in einem Textdokument und beziehen sich auf die Position im Text. Hier wird die Position des Startpunktes anhand der Seite auf der die Kodierung startet (`page1`), und der Position als Kombination aus dem Index des Paragraphen (`startKey`) und dem Index des Zeichens im Text (`pos1`) definiert. Der Endpunkt wird analog in `page2`, `endKey` und `pos2` gespeichert.
- **Bereichskodierungen:** Da PDFs neben Texten auch Grafiken beinhalten können, bieten Bereichskodierungen die Möglichkeit, einen beliebigen rechteckigen Bereich zu kodieren. Dieser Bereich kann Seitengrenzen nicht überschreiten. Entsprechend wird eine Bereichskodierung über die Seite (`page`), die Position der oberen linken Ecke des Bereiches (`x`, `y`) und die Höhe und Breite (`height`, `width`) des markierten Bereichs repräsentiert (*Listing 4.5*). Bereichskodierungen besitzen zudem aktuell noch keine Preview des kodierten Bereichs.

## 4. Architektur

---

```
{
  "Key": string ,
  "ID/Name": int ,
  "DISCRIMINATOR": string , // Voll qualifizierender JAVA class name
  "author": string ,
  "codesystemID": int ,
  "documentId": int ,
  "endKey": int ,
  "localCodeId": int ,
  "page1": int ,
  "page2": int ,
  "pos1": int ,
  "pos2": int ,
  "preview": string ,
  "projectId": int ,
  "projectType": ProjectTypeEnum ,
  "startKey": int ,
  "timestamp": DateTime
}
```

---

**Listing 4.4:** Repräsentation der ursprünglichen Kodierungsdaten-Struktur für PDF-Textkodierungen als annotiertes JSON

---

```
{
  "Key": string ,
  "ID/Name": int ,
  "DISCRIMINATOR": string , // Voll qualifizierender JAVA class name
  "author": string ,
  "codesystemID": int ,
  "documentId": int ,
  "height": float ,
  "localCodeId": int ,
  "page": int ,
  "projectId": int ,
  "projectType": ProjectTypeEnum ,
  "timestamp": DateTime
  "width": float ,
  "x": float ,
  "y": float ,
}
```

---

**Listing 4.5:** Repräsentation der ursprünglichen Kodierungsdaten-Struktur für PDF-Bereichskodierungen als annotiertes JSON

## 4.2 Aktuelle Architektur

### 4.2.1 Hauptkomponenten

Nach der Einführung des kollaborativen Editierens ist die Grundstruktur um die CES-Funktionalität erweitert. Anders als in *Abbildung 4.3* dargestellt, handelt es sich bei RTCS und CES jedoch noch um zwei eigenständige Node-basierte Server und nicht um eine einzige Serverinstanz mit kombinierter Funktionalität. Dies ist allerdings ein rein technisches Detail und macht für die Betrachtung der Architektur keinen Unterschied, da die beiden Server grundsätzlich problemlos in einen einzigen überführt werden könnten.

Der Hauptunterschied zum ursprünglichen Datenfluss liegt darin, dass sowohl die Coding-Daten, als auch die Dokumente selbst über den CES mittels Websocket-Verbindungen übertragen werden.

Der CES erstellt dabei yDoc-Dokumente, die zur Synchronisation der Daten dienen. Es wird ein Dokument pro Codesystem angelegt, welches von allen Nutzern in der selben Projektversion genutzt wird. Zudem wird für jedes in QDacity geöffnete Dokument ein yDoc angelegt, das alle Änderungen am Dokument in Echtzeit zwischen den Bearbeitern teilt.

### 4.2.2 Datenstruktur

Die Daten für die Dokumente werden hierbei als yDoc in den Google Cloud Buckets abgelegt, was zur aktualisierten Speicherstruktur in *Abbildung 4.4* führt. Hier wird zwar noch eine HTML-Datei für das Dokument angelegt, jedoch war diese zu Beginn dieser Arbeit nicht mit Daten befüllt. Zur einfacheren Typ-Unterscheidung wurde der neue Dokumenttyp COLLABORATIVETEXT eingeführt. Dies ermöglicht es, zwischen bestehenden TEXT-Dokumenten in HTML-Struktur und neuen Yjs-basierten Daten zu unterscheiden. So werden ungültige Anfragen verhindert, da nie der falsche Dokumenttyp, und damit eine nicht vorhandene Datei, angefordert wird.

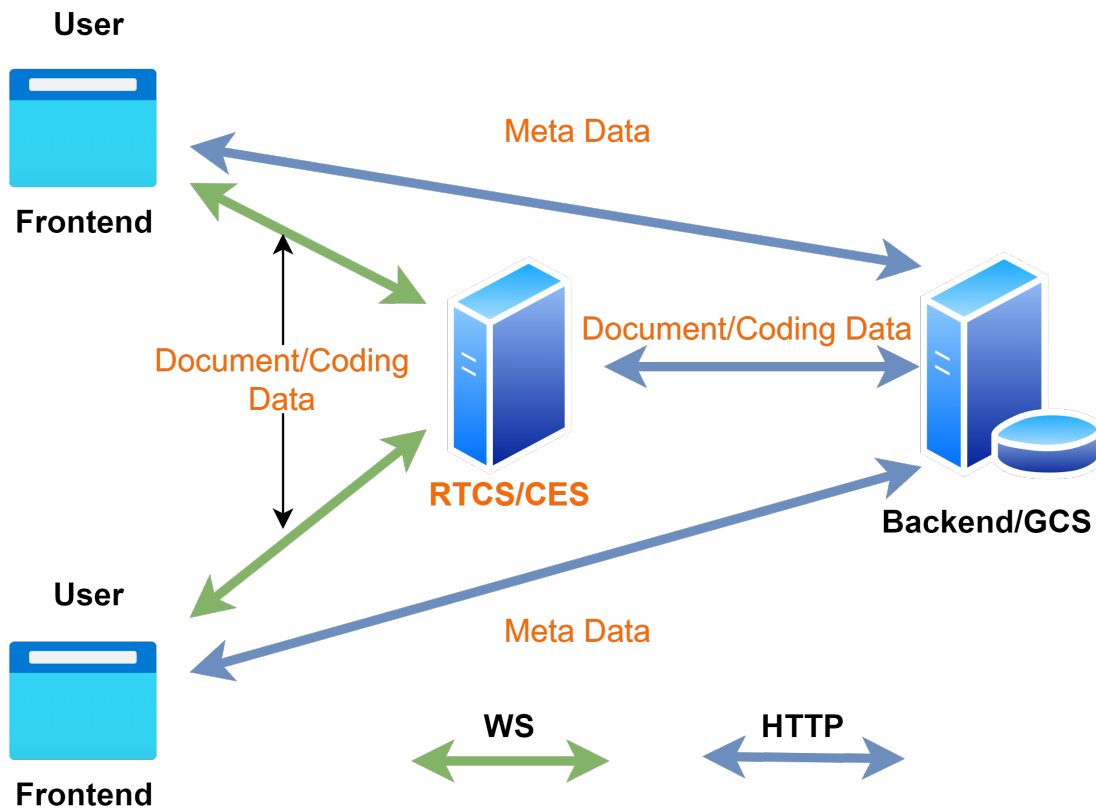


Abbildung 4.3: Aktuelle QDAcity-Architektur (Dürsch, 2023, S. 47)

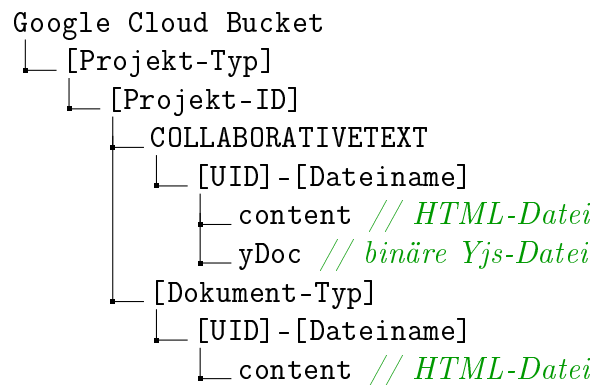


Abbildung 4.4: Aktuelle Bucket-Struktur eines QDAcity-Buckets

Die Kodierungsdaten werden vom CES transient verwaltet. Die eigentliche Datenhaltung erfolgt wie vor Einführung des kollaborativen Editierens im Datastore. Wenn das Frontend ein CS vom CES anfordert, so lädt dieser alle notwendigen Entitäten über das Backend aus dem Datastore. Dann wird damit ein yDoc befüllt, welches zur Synchronisation mit den einzelnen Clients verwendet wird. Alle

darin eingespielten Änderungen werden in Intervallen und beim Schließen des CS über das Backend in den Datastore zurückgeschrieben. So beinhaltet dieser alle Daten, die für ein späteres Laden des CS oder eine Analyse auf den Codes und Kodierungen notwendig sind.

### 4.2.3 Datenmigration

Da sich das Speicherformat der Daten geändert hat, ergibt sich die Notwendigkeit einer geeigneten Strategie zur Datenmigration. Da das kollaborative Editieren den neuen Standard darstellt und den bisherigen Mechanismus ablösen soll, wurde sich hier für eine einmalige Migration der Daten entschieden.

Diese Migration unterscheidet sich in ihrem Mechanismus von den bisher innerhalb von QDAcity verwendeten Migrationen, da sie nicht direkt vom Backend durchgeführt werden kann. Alle früheren Migrationen innerhalb von QDAcity konnten als Task für das Backend implementiert werden. Da das Backend jedoch keine Yjs-Daten schreiben kann, muss diese Portierung zwingend entweder im CES oder im Frontend durchgeführt werden.

Um bei zeitlich nahem Zugriff keine doppelte Konvertierung zu starten und entsprechend Kollisionen beim Speichern der initial konvertierten Dokumente zu verhindern, fiel die Wahl hierbei auf den CES. Wird ein TEXT-Dokument zum ersten mal von einem Nutzer über den CES angefordert, so lädt dieser zunächst die content-HTML-Datei. Aus jener werden alle codingKey-Attribute entfernt und der bestehende Text wird in ein yDoc geladen, welches als neue COLLABORATIVE-TEXT-Datei mit einer Kopie der content-Datei abgelegt wird. Die TEXT-Datei wird nach Abschluss der Migration gelöscht.

Bei dieser Migration gehen alle Kodierungen im Dokument verloren, weshalb dieser Stand des CES nie in die Produktionsumgebung deployed wurde.

## 4.3 Geplante Architektur

Abschließend soll nun die im Rahmen dieser Arbeit geplante Architektur beschrieben werden.

### 4.3.1 Hauptkomponenten

In der (im Rahmen dieser Arbeit) finalen Architektur, soll QDAcity wieder aus drei Hauptkomponenten bestehen, wobei der CES mit dem RTCS zusammengeführt wird. Dieser wird dann wieder den Namen RTCS tragen, wobei er im folgenden zur klareren Unterscheidung weiterhin als CES bezeichnet wird. Dadurch ergibt sich die in *Abbildung 4.5* dargestellte aktualisierte Architektur. Dabei ist vor allem auch der geänderte Datenfluss der Dokumente und Kodierungsdaten zu beachten. Dafür ist nicht länger das Backend führend, die Daten werden stattdessen direkt

vom CES verwaltet. Dieser liefert die Informationen nur noch unidirektional ans Backend, um dort Datenanalysen zu ermöglichen.

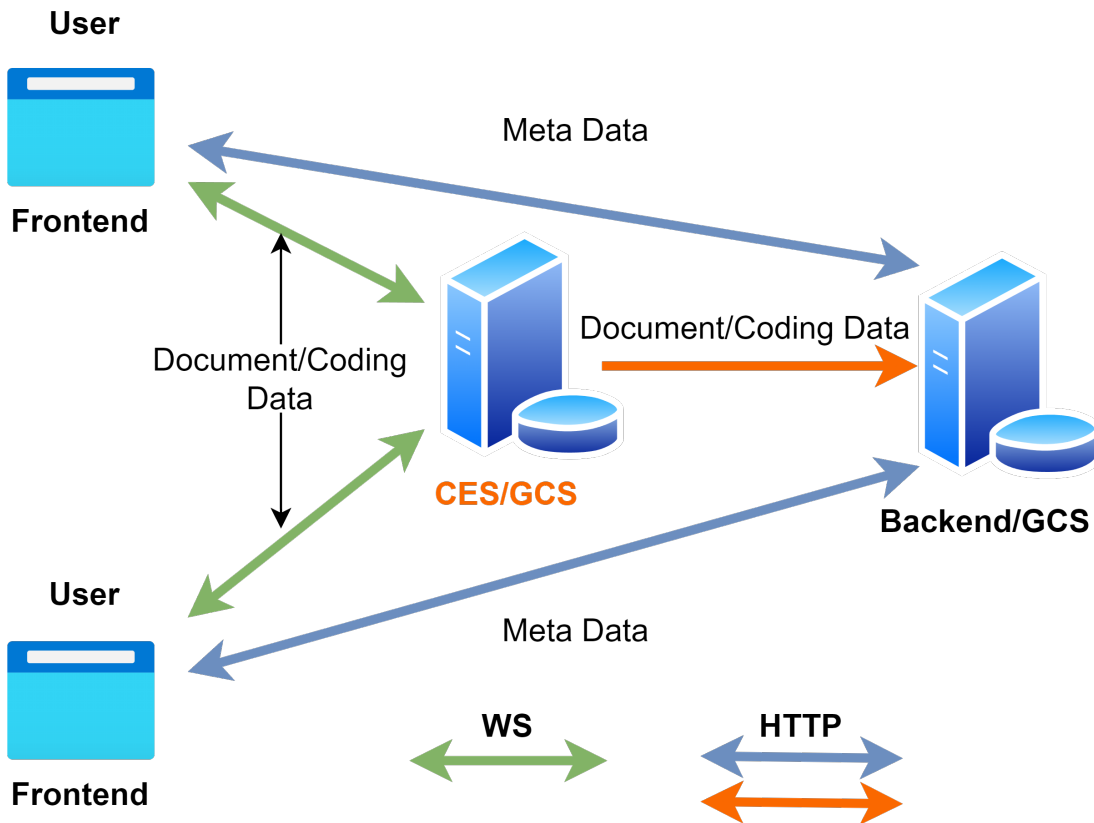


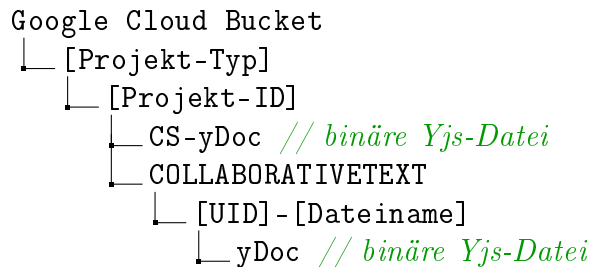
Abbildung 4.5: Geplante QDacity-Architektur

### 4.3.2 Datenstruktur

Die Daten für die Dokumente werden weiterhin als yDoc in den Google Cloud Buckets abgelegt (siehe *Abbildung 4.6*). Hier wird keine HTML-Datei mehr für das Dokument angelegt und alte TEXT-Dokumente werden bei der ersten Verwendung in COLLABORATIVETEXT-Dokumente umgewandelt, weshalb diese auch nicht mehr in der Bucket-Struktur repräsentiert werden.

Zudem werden die Kodierungsdaten nicht länger im Datastore abgelegt, sondern werden, wie auch die Dokumente, als yDoc in Google Cloud Buckets gespeichert. Dies ermöglicht eine effizientere Nutzung des CS, da dieses nicht bei jeder Anforderung aus dem Datastore geladen und in ein yDoc umgewandelt werden muss, sondern direkt in seiner nativen Form gespeichert werden kann. Da das CS jeweils fest einem Projekt zugeordnet ist, findet sich dieses im jeweiligen Projektordner neben den Ordnern für die einzelnen Dokumententypen (*Abbildung 4.6*). Auch in

EXERCISE- und REVISIONs-Projekten liegt hier ein eigenes CS-yDoc vor, da damit die Unveränderlichkeit sichergestellt werden kann. Eine vollständigere Repräsentation findet sich im Anhang in *Abbildung 3*.



**Abbildung 4.6:** Geplante Bucket-Struktur eines QDAcity-Buckets

Mit der Portierung des CS soll auch die Speicherung der Kodierungen für TEXT-Dokumente an das neue Schema angepasst werden. Diese werden bisher, wie in 4.1.2 beschrieben, als Eintrag im Datastore angelegt, welcher bei jeder Verschiebung des Indexes von Anfangs- bzw. Endposition aktualisiert werden muss. Yjs erlaubt es, mehrere Änderungen zu einer Transaktion zusammenzufassen, dies ist jedoch nur innerhalb einer Yjs-Datei möglich. Da die Kodierungen aber innerhalb von QDAcity logisch dem CS zugeordnet sind, werden sie im CS-yDoc verortet.

Bei Beibehaltung der alten Datenstruktur müssten folglich bei einer Anpassung innerhalb eines Codings Änderungen sowohl im CS-yDoc, als auch im yDoc des Dokuments gleichzeitig vorgenommen werden. Sollte eines dieser Updates nicht korrekt propagiert werden, geraten die Informationen über die Kodierung in einen invaliden Zustand.

Um dem entgegenzuwirken sollen die Kodierungspositionen als `stored positions`<sup>8</sup> im Text-yDoc hinterlegt werden. Dies hat außerdem den Vorteil, dass der für QDAcity individuell entwickelte Mechanismus, der durch eine Erweiterung in den Slate-Editor integriert wurde, zum Updaten der Kodierungspositionen nicht weiter in diesem Umfang benötigt wird.

Hierbei soll jeweils eine `stored position` den Start- und den Endpunkt markieren. Dabei folgen sie dem Benennungsschema `[CodingId].start` und `[CodingId].end` zur einfachen und eindeutigen Referenzierung.

Für die im CS-yDoc persistierten Kodierungen ergibt sich daraus die in *Listing 4.6* dargestellte neue Struktur.

<sup>8</sup> <https://docs.slate-yjs.dev/concepts/stored-positions>

---

```
{
  "id": int ,
  "DISCRIMINATOR": string , // Voll qualifizierender JAVA class name
  "author": string ,
  "documentId": int ,
  "localCodeId": int ,
  "preview": string ,
  "timestamp": DateTime
}
```

---

**Listing 4.6:** Repräsentation der geplanten Kodierungsdaten-Struktur als annotiertes JSON

Durch die angepasste Dokumentstruktur muss auch der Prozess zum Update der Preview angepasst werden. Dabei kann der bestehende Algorithmus nur in Teilen übernommen werden, da dieser auf der Verwendung der `CodingKey`-Struktur basiert. Die Migration wird in *5.1.1* genauer beschrieben.

Die Aktualisierung der Preview einer Kodierung muss hier weiterhin im `CS-yDoc` durchgeführt werden. Sollte diese jedoch nicht zu jedem Zeitpunkt mit der aktuellen Version einer Kodierung übereinstimmen, ist die Auswirkung hier eher zu vernachlässigen, als bei der eigentlichen Kodierungsposition.

Anders ist es um die Kodierungen innerhalb von `PDF`-Dokumenten gestellt. Da diese aufgrund der statischen Natur von `PDFs` nicht durch Modifikationen am zugrundeliegenden Dokument beeinflusst werden können, muss hier die Position wie schon bisher nicht im Dokument hinterlegt werden.

Entsprechend bleibt hier die ursprüngliche Struktur, wie bereits in *Listing 4.4* und *Listing 4.5* dargestellt, nahezu erhalten. Lediglich nicht länger benötigte Eigenschaften (`codesystemId`, `ID/Name`, `projectId`, `projectType`) werden entfernt und die Kodierungen werden, wie in *Listing 4.8* und *Listing 4.7* dargestellt, im `CS-yDoc` hinterlegt.

---

```
{
  "id": int ,
  "DISCRIMINATOR": string , // Voll qualifizierender JAVA class name
  "author": string ,
  "documentId": int ,
  "height": float ,
  "localCodeId": int ,
  "page": int ,
  "timestamp": DateTime
  "width": float ,
  "x": float ,
  "y": float ,
}
```

---

**Listing 4.7:** Repräsentation der geplanten Kodierungsdaten-Struktur für `PDF`-Bereichskodierungen als annotiertes JSON



---

```

{
  "id": int ,
  "DISCRIMINATOR": string , // Voll qualifizierender JAVA class name
  "author": string ,
  "documentId": int ,
  "endKey": int ,
  "localCodeId": int ,
  "page1": int ,
  "page2": int ,
  "pos1": int ,
  "pos2": int ,
  "preview": string ,
  "startKey": int ,
  "timestamp": DateTime
}

```

---

**Listing 4.8:** Repräsentation der geplanten Kodierungsdaten-Struktur für PDF-Textkodierungen als annotiertes JSON

### 4.3.3 Datenmigration

Da die grundlegende Dokumentstruktur im Rahmen dieser Arbeit nicht weiter verändert wird, kann die bestehende Migrationsstrategie beibehalten werden. Diese muss jedoch weiter verbessert werden, sodass auch eine Migration der Kodierungen in bestehenden Dokumente durchgeführt wird.

Dafür können zwei mögliche Vorgehensweisen in Betracht gezogen werden, die im Folgenden kurz beschrieben sind:

Zum einen bietet sich die Erweiterung der bestehenden Migration im CES an. Dabei muss die Position einer Kodierung beim Entfernen gecacht werden. Im Folgenden muss dann die entsprechende Position der Kodierung in der resultierenden Datei gefunden und als `stored positions` im `yDoc` persistiert werden. Die `stored positions` sind jedoch in der `slate-yjs`-Dokumentation<sup>9</sup> nur unzureichend dokumentiert. Dadurch muss für ihre Erstellung entweder der Bibliothekscode analysiert werden, um diese Umformung selbst vornehmen zu können, oder `slate-yjs` als weitere Bibliothek in den CES integriert werden. Da die NodeJS-API jedoch von der normalen Browser-API abweicht, kann sich dies möglicherweise als nicht realisierbar herausstellen.

Als alternative Vorgehensweise bleibt in diesem Fall nur, die Migration der Kodierungen in das Frontend zu übertragen. Hier kann auf die volle `slate-yjs` Bibliothek zugegriffen werden, was die Migration der Kodierungen ermöglicht. In diesem Fall muss jedoch das ursprüngliche HTML-Dokument in das `yDoc` übertragen und mit an das Frontend ausgespielt werden.

---

<sup>9</sup> <https://docs.slate-yjs.dev/concepts/stored-positions>

Abschließend muss auch das gesamte CS aus dem Datastore extrahiert und in einem yDoc abgelegt werden. Dies kann jedoch durch eine Erweiterung der Migrationsfunktion des CES erfolgen.

### 4.4 Notwendigkeit einer neuen Schnittstelle

Obwohl die Daten des CS vom CES nun führend in einem eigenen yDoc gespeichert werden, müssen Änderungen auch im Datastore aktuell gehalten werden, um die Erzeugung von ICA-Berichten zu ermöglichen. Da diese doppelte Speicherung anfällig für Inkonsistenzen ist und unnötige Kosten verursacht, soll dieser Zustand möglichst schnell behoben werden. Dafür soll der CES um eine eigene Schnittstelle erweitert werden, welche es ermöglicht, dass die von Anavatti (2022) erweiterten und eingeführten Auswertungen mit den kollaborativen Daten arbeiten können. Dabei ist zudem zu beachten, dass Dürsch (2023) bei der Einführung der Echtzeit-Kollaboration die Datenstruktur der Text-Dateien wie in 4.2.3 beschrieben grundlegend ändern musste. Diese Änderungen waren notwendig, da es andernfalls schwer bis nicht möglich gewesen wäre, Änderungen die zur gleichen Zeit im selben Textabschnitt gemacht werden, konfliktfrei aufzulösen (Dürsch, 2023). Jedoch verhindert auch diese Umstellung die Erstellung der ICA-Berichte, da die hierfür verwendete Analyselogik von den bisherigen Datenstrukturen Gebrauch macht, welche im neuen System nicht mehr verfügbar sind. Die zur Verarbeitung natürlicher Sprache (engl. *Natural Language Processing*) (NLP) vorausgesetzten Analysen Bibliotheken stehen für JavaScript nicht im gleichen Umfang bereit, weshalb diese Logik nicht ohne Weiteres in den CES umgezogen werden kann.

Da die Kodierung im Rahmen der QDA stark der subjektiven Einschätzung des Kodierers unterliegt, ermöglichen die ICA die Übereinstimmung zwischen Kodierenden automatisiert auszuwerten. Dadurch kann ohne großen Aufwand ermittelt werden, ob ein Text auch bei Reduzierung der subjektiven Einflüsse (durch den Vergleich mehrerer subjektiver Auswertungen) ähnlich kodiert wurde.

Folglich nehmen die ICA-Berichte gerade bei kollaborativer Verwendung von QDAcity eine wichtige Rolle ein, weswegen hier ein Weg gefunden werden musste, um die Berichtserstellung wieder zu ermöglichen. Besonderes Augenmerk lag dabei auf einer mit möglichst geringen Aufwänden belastete Umsetzung, da diese die Einführung des überarbeiteten kollaborativen Editierens weiter verzögern würde.

Eine einfache Umstellung dieser Auswertungen auf die von Yjs verwendete Datenstruktur ist hierbei nicht ohne Weiteres möglich, da diese Bibliothek ihrem Namen nach in JavaScript implementiert wurde und es aktuell keine Java-Implementierung dieser Bibliothek gibt. Somit müsste hier zunächst ein Java-Interpreter für die binären Yjs-Dateien entwickelt werden, was einen stark erhöhten Entwicklungsaufwand zur Folge hätte.

Stattdessen soll mit dem gewählten Weg einer neuen Schnittstelle die effizientes-

te Schließung der Lücke zwischen diesen beiden Komponenten verfolgt werden. Dabei werden im RTCS die Texte nativ in JavaScript aus den Yjs-Dateien ausgelesen und in das klassische HTML-Format (siehe *Listing 4.1*) überführt. Dieses kann als Datenpaket zusammen mit dem verwendeten CS und den Kodierungsdefinitionen auf Anforderung des Backends generiert und ausgespielt werden.

## 4.5 Mechanismen für sichere Server-Server-Kommunikation

Teil dieser Arbeit ist, die REST-API derart abzusichern, dass nur QDAcity-Server Zugriff auf die vom RTCS verwalteten Daten bekommen. Im Folgenden sollen einige Möglichkeiten zur Authentisierung anderer Server gegenüber der REST-API vorgestellt und diskutiert werden.

### 4.5.1 JSON Web Token

Gemäß Lesavre et al. (2021) sind JavaScript Object Notation Web Tokens (JWTs) die am meisten verwendete Form von eigenständigen Tokens zur Authentifizierung, die digital zertifizierte oder verschlüsselte Daten enthalten können. Dementsprechend findet man für alle gängigen Programmiersprachen Bibliotheken, welche die einfache Einbindung von JWTs in eine Anwendung ermöglichen (auth0 by Okta, n. d.).

Ein weiterer Vorteil von JWTs ist, dass diese bereits aktuell von QDAcity zur Nutzerauthentisierung verwendet werden. Dadurch müsste keine zusätzliche Technologie eingeführt werden, was die sowieso schon komplexe QDAcity-Infrastruktur unnötig unübersichtlich gestalten würde.

Im Folgenden werden drei mögliche Übertragungsarten für das JWT vorgestellt, wie sie von Jones und Hardt (2012) definiert werden. Dieser Request for Comments (RFC) bezieht sich auf die Nutzung von Bearer Tokens in OAuth, wobei laut Jones et al. (2015) JWTs als OAuth Bearer Token verwendet werden können.

#### Authorization Header

Das JWT kann mittels des **Authorization**-Hypertext Transfer Protocol (HTTP)-Headers bei einer Serveranfrage mitgeschickt werden, wobei der Header dem Schema **Authorization: Bearer <Token>** entspricht und das Token Base64-encodiert wird.

Dies ist die empfohlene Übertragungsart für Bearer Token (Jones und Hardt, 2012).

### POST Body

Eine mögliche Alternative zur Übertragung im Header ist die Einbettung in den Body einer POST-Anfrage, wobei das Token als Parameter nach dem Schema `access_token=<token>` in eine URL-encodierte Nachricht (Content-Type: `application/x-www-form-urlencoded`) eingefügt wird.

Diese Verwendung wird jedoch nur empfohlen, wenn keine Möglichkeit besteht, den `Authorization`-Header zu verwenden (Jones und Hardt, 2012).

### Query-Parameter

Als dritte Möglichkeit wird zudem genannt, dass das Token auch mittels eines URI Query Parameters in Form von `https://qdacity.com/resource?access_token=<token>` versendet werden könnte.

Jedoch kann hierbei nicht ausgeschlossen werden, dass das Token geloggt wird und im Folgenden nicht mehr als geheim angesehen werden könnte. Daher wird von dieser Methode abgeraten, sie sollte nur verwendet werden, wenn keine der zuvor genannten Varianten angewendet werden kann (Jones und Hardt, 2012).

### 4.5.2 mTLS

Bei der geplanten Verwendung der API in `QDacity` sollen keine einzelnen Nutzer sondern lediglich Server-Identitäten geprüft werden, um zu verhindern das keine Daten an unbefugte Server/Clients gesendet werden. Dadurch gibt es neben der Authentisierung durch ein JWT auch die Möglichkeit, dass sich die einzelnen Server über Zertifikate und die Nutzung von mutual Transport Layer Security (mTLS) authentisieren.

Bei mTLS handelt es sich um eine Erweiterung von Transport Layer Security (TLS), bei der nicht nur das X.509-Zertifikat des Servers zur Prüfung dessen Identität herangezogen wird, sondern auch das Zertifikat des Clients überprüft wird. Dadurch ist es möglich, zwei Server von einer vertrauenswürdigen Certificate Authority (CA) authentifizieren zu lassen. Diese Server können sich dann gegenseitig authentisieren, ohne, dass einer um die Existenz des jeweiligen anderen Servers wissen muss (Jones und Hardt, 2012).

Dabei wird der TLS-Handshake, im Folgenden fett markiert, erweitert (Cloudflare, n. d.):

- Client initiiert Verbindung zum Server
- Server sendet TLS-Zertifikat an Client
- Client verifiziert das Server-Zertifikat
- **Client sendet sein TLS-Zertifikat an Server**
- **Server verifiziert das Client-Zertifikat**

- **Server gewährt Zugang**
- Client und Server tauschen Informationen über TLS-verschlüsselte Verbindung aus

Die Erweiterung ist auch in *Abbildung 4.8* dargestellt, wobei *Abbildung 4.7* den standardmäßigen TLS-Handshake repräsentiert.

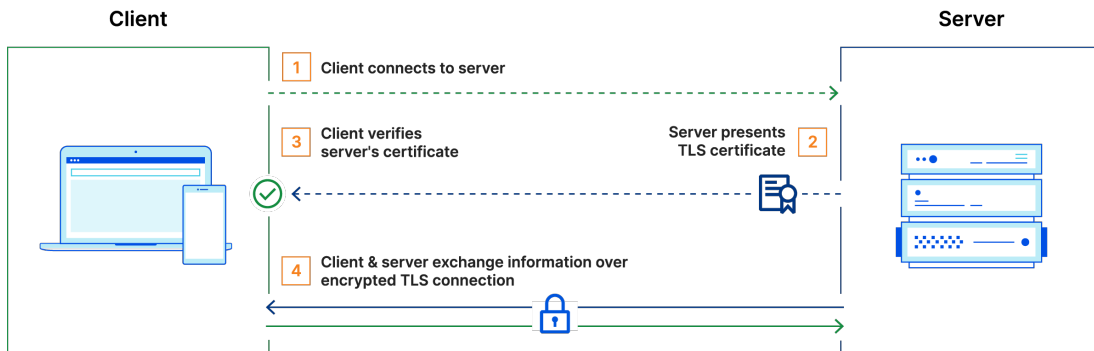


Abbildung 4.7: Normaler Ablauf TLS-Handshake<sup>10</sup>

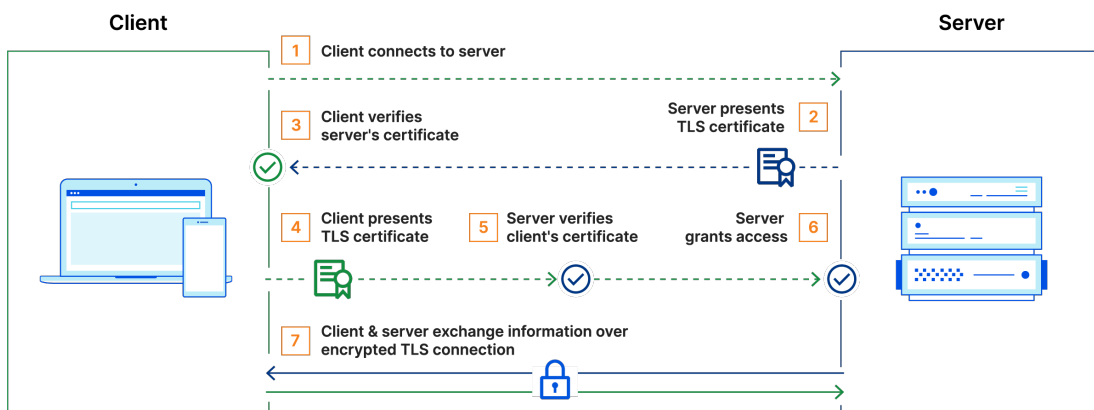


Abbildung 4.8: Erweiterter Ablauf mTLS-Handshake<sup>11</sup>

Dieses Vorgehen bietet zwei wichtige Vorteile:

Zum einen kann auf die Verwendung einer JWT-Bibliothek verzichtet werden, da der QDAcity RTCS JWTs nicht direkt verarbeiten muss. Aktuell verwendet der RTCS zwar JWTs, diese werden jedoch nicht geprüft sondern dienen nur dazu, den Nutzer am Backend zu authentisieren, daher ist dafür aktuell auch keine

<sup>10</sup> [https://www.cloudflare.com/resources/images/slt3lc6tev37/37w1tzGsD4XvYUkQCHbWG8/6fbbb48d0f5077cc2c662a4cc6817b1c/how\\_tls\\_works-what\\_is\\_mutual\\_tls.png](https://www.cloudflare.com/resources/images/slt3lc6tev37/37w1tzGsD4XvYUkQCHbWG8/6fbbb48d0f5077cc2c662a4cc6817b1c/how_tls_works-what_is_mutual_tls.png)

<sup>11</sup> [https://www.cloudflare.com/resources/images/slt3lc6tev37/5SjaQfZzDLEGqyzFkA0AA4/d227a26bbd7bc6d24363e9b9aaabef55/how\\_mtls\\_works-what\\_is\\_mutual\\_tls.png](https://www.cloudflare.com/resources/images/slt3lc6tev37/5SjaQfZzDLEGqyzFkA0AA4/d227a26bbd7bc6d24363e9b9aaabef55/how_mtls_works-what_is_mutual_tls.png)

JWT-Bibliothek eingebunden. Da alle QDAcity Komponenten jedoch eine API-Schnittstelle haben, kann mTLS direkt über die jeweilige Server-/ bzw. Request-Komponente verwendet werden, weil diese mTLS bereits unterstützen.

Außerdem reduziert die Verwendung von mTLS den notwendigen Aufwand bei der Implementierung, da die Autorisierung auf eine tiefere Ebene verlagert wird. Das lässt sich am einfachsten am klassischen Open Systems Interconnection (OSI)-Schichtenmodell darstellen. Dabei ist jedoch zu beachten, dass dieses Modell eigentlich eine unzureichende Repräsentation für den TCP/IP-Netzwerkverkehr ist und daher hier nur zur leichteren Veranschaulichung verwendet wird. Im eigentlich passenderen TCP/IP-Modell sind die Schichten 5-7 zusammengenommen, wodurch die hier verwendete Veranschaulichung nicht zum Tragen käme (VaibhavBilotia et al., 2022).

Wie in *Abbildung 4.9* dargestellt, besteht das OSI-Schichtenmodell aus 7 Schichten, wobei durch mTLS die Autorisierung von Schicht 7 (der Anwendungsschicht) auf Schicht 5 (die Kommunikationsschicht) herabgesetzt wird (Elektronik Kompendium, n. d.). Dadurch wird die notwendige Implementierungsarbeit reduziert, da diese bei der Anwendungsschicht zumeist Erweiterungen der eigenen oder eingesetzten Bibliotheken erfordert. Bei den tiefer liegenden Schichten kann die Anpassungen hingegen durch reine Konfiguration (oder im schlimmsten Fall einen Wechsel) der verwendeten Bibliotheken erfolgen.

Für die Schnittstelle des CES wird die Verwendung von mTLS empfohlen.

OSI Modell		
Schicht	Name	Beispielhafte Protokolle
7	Anwendungsschicht	HTTP, FTP, SMTP, POP, IMAP,
6	Darstellungsschicht	...
5	Kommunikationsschicht	TLS
4	Transportschicht	TCP / UDP
3	Vermittlungsschicht	IPv4 / IPv6
2	Sicherungsschicht	IEEE 802.3 (Ethernet),
1	Bitübertragungsschicht	IEEE 802.11 (WLAN), ...

**Abbildung 4.9:** OSI-Schichtenmodell mit Protokollen (Elektronik Kompendium, n. d.)

# 5 Design und Implementierung

In diesem Kapitel werden alle Änderungen und die dabei relevanten Designentscheidungen dargelegt. Dabei wird zunächst auf die grundlegende Anbindung des Frontends eingegangen, bevor die Einbindung weiterer Features in den CES beschrieben wird. Folgend wird ein neuer Migrationsmechanismus zur Übernahme der Kodierungen beschrieben, bevor abschließend notwendige Änderungen am CES erläutert werden.

## 5.1 Implementierung kollaborativer Features mit Yjs

Die wichtigste Anpassung war die Anbindung des CES an den Client und damit einhergehend die Einführung des kollaborativen Editierens im Frontend. Die dafür notwendigen Anpassungen werden folgend beschrieben.

### 5.1.1 Verwendung kollaborativer Daten im Frontend

Zunächst lag das Hauptaugenmerk auf der vollständigen Einführung des kollaborativen Editierens mittels Yjs im Frontend. Hierfür waren vor Beginn dieser Arbeit lediglich zwei Hocuspocus<sup>1</sup>-Provider im Frontend erstellt worden. Einer im Editor für Textdokumente, welcher zum Laden dieser Textdokumente vom CES und zur Anzeige derselben innerhalb eines Slate-Editors dient. Der Zweite als eigene Komponente namens `CollaborativeCodesystemProvider`, welche die benötigten Codesystemdaten mit dem CES synchronisiert.

Im Folgenden werden die einzelnen integrierten Komponenten kurz aufgelistet.

---

<sup>1</sup> <https://tiptap.dev/hocuspocus/introduction>

### CodeSystem

Nach der Vorbereitung durch Dürsch (2023) konnte das CS als erste Komponente integriert werden.

Dafür wurde je eine kollaborative Kopie der betroffenen Komponenten als `CodesystemCollaborative`, `CodesystemToolbarCollaborative` und `CodesystemWithCheckboxCollaborative` angelegt.

In diesen Dateien mussten alle Referenzen auf den `SyncService` (Frontend Komponente zur Verknüpfung des RTCS) durch entsprechende Verwendungen des `CollaborativeCodesystemProviders` ersetzt werden.

Dabei musste die CS-Toolbar angepasst werden, sodass durch den Nutzer angeforderte Änderungen nicht an den `SyncService`, sondern an den `Hocuspocus-Provider` übergeben wurden.

### Textdokumente

Auch für die Textdokumente war, wie bereits beschrieben, die grundsätzliche Anbindung vorhanden. Dabei blieb jedoch durch die fehlende Integration des CS die Anzeige der Kodierungen aus.

Hierfür wurden, wie in 4.3.2 dargelegt, `stored positions` verwendet, um die Positionen einer Kodierung innerhalb der Textdatei zu speichern. Dafür musste entsprechend die Anzeige der Positionen innerhalb der `CodingBrackets` angepasst werden, sodass diese nicht die an der Kodierungsdefinition befindlichen, sondern die im Dokument hinterlegten Positionen verwendet.

Außerdem musste auch der zweite Editormodus bei Textdokumenten, der Kodierungsmodus, angepasst werden. Dieser war zuvor in `QDAcity` als eigene Editorerweiterung definiert. Dabei wurde die `apply()`-Funktion des `Slate-Editors` überschrieben, sodass im Kodierungsmodus nur `set_selection`-Operationen erlaubt wurden, um dem Nutzer die notwendige Selektierung von Textpassagen zu ermöglichen.

Die überarbeitete Überschreibung der `apply()`-Funktion ist in *Listing 5.1* wiedergegeben. Hier ist im Vergleich zur ursprünglichen Implementierung zusätzlich auch `merge_node` als erlaubte Operation im Kodierungseditor (`readOnly == true`) aufgenommen (Z. 1). Dies ist notwendig, da `Yjs` externe Operationen teilweise mit einer lokal erstellten `merge_node`-Operation anwendet.

Zunächst wird geprüft, ob aktuell der Kodierungsmodus aktiviert ist. Ist dies der Fall, so wird die Anwendung der Operation abgebrochen, wenn die Änderung lokal erstellt wurde und nicht zu den erlaubten Operationen zählt (Z. 5f). Handelt es sich um eine `remove_text`-Operation, so wird zudem das `ignore_next_set_selection`-Flag auf `true` gesetzt (Z. 7f). Dies ist notwendig, da `Slate` nach dem Entfernen des Textes eine `set_selection`-Operation anwendet, um entsprechend auf den geänderten Index der Cursorposition durch das Entfernen des Textes zu reagieren. Wird die Textänderung nicht angewendet, so würde der Cursor im Text



bewegt werden, ohne, dass ein Grund ersichtlich wäre.

Zudem werden auch `merge_node`-Operationen, die auf `paragraph`-Ebene angewendet werden sollen, übersprungen (Z. 11ff), da diese nur durch lokale Nutzereingaben erzeugt werden können.

Folgend wird noch überprüft, ob es sich um eine `set_selection`-Operation handelt, die übersprungen werden soll (Z. 15ff).

Abschließend wird die Operation angewandt, insofern die Ausführung nicht auf Grund der zuvor genannten Punkte abgebrochen wurde (Z. 20).

---

```

1 const readOnlyOperations = [ 'set_selection', 'merge_node' ];
2 const { apply } = editor;
3 let ignore_next_set_selection = false;
4 editor.apply = (operation) => {
5   if (readOnly && YjsEditor.isLocal(editor)
6     && !readOnlyOperations.includes(operation.type)) {
7     if (operation.type === 'remove_text')
8       ignore_next_set_selection = true;
9     return;
10  }
11  if (readOnly && operation.properties?.type === 'paragraph') {
12    ignore_next_set_selection = true;
13    return;
14  }
15  if (ignore_next_set_selection
16    && operation.type === 'set_selection') {
17    ignore_next_set_selection = false;
18    return;
19  }
20  apply(operation);
21 };

```

---

**Listing 5.1:** Handling von Änderungen

Durch die Anpassung der Editorerweiterung wurde auch der bisher verwendete Algorithmus zum Update der Kodierungspreview entfernt, da dieser auf die `CodingKey`-Struktur angewiesen war.

Dieser wurde durch eine Funktion im `Texteditor` ersetzt, welche mittels des `Yjs`-Observers `observeDeep` bei lokalen Änderungen am Textdokument aufgerufen wird.

Dabei ist die Funktion, wie in *Listing 5.2* dargestellt, in zwei funktionale Abschnitte gegliedert. Im ersten Teil (Z. 1-19) wird zunächst über alle für das Dokument relevanten Kodierungen iteriert und die aktuelle Version der Preview anhand der `stored positions` abgeleitet. Entspricht diese nicht der aktuell an der Kodierung gespeicherten Version, so wird sie zum Update vorgesehen.

Im zweiten Teil (Z. 21-31) werden die zum Update vorgesehenen Kodierungen aktualisiert. Dabei wird unterschieden, ob die berechnete Preview einen Wert

enthält oder leer ist:

Enthält die Preview Daten, so wird die aktualisierte Kodierung in den Yjs-Provider aufgenommen und an andere Clients propagiert.

Bei einer leeren Preview wird die Kodierung gelöscht, da ihr Text durch Bearbeitung aus dem Dokument entfernt wurde. Dabei werden auch die `stored positions` aus dem Dokument entfernt.

---

```
1 const updateCodingPreviews = (codeSystem, codings, editor,
  ↪ documentId) => {
2 const updatedCodings = [];
3 const documentCodings = codings.filter((c) =>
  c.documentId == documentId);
4
5 documentCodings.forEach((coding) => {
6 const start = YjsEditor.position(editor, `${coding.id}.start`);
7 const end = YjsEditor.position(editor, `${coding.id}.end`);
8 if (start && end) {
9 const range = Editor.range(editor, start, end);
10 const domRange = CodingPlugin.toDOMRange(editor, range);
11 const preview = domRange.toString();
12 if (preview != coding.preview) {
13     updatedCodings.push({
14         ... coding,
15         preview: preview,
16     });
17 }
18 }
19 });
20
21 updatedCodings.forEach((updatedCoding) => {
22 if (updatedCoding.preview.length > 0) {
23     codeSystem.setCoding(updatedCoding);
24 } else {
25     codeSystem.removeCoding(updatedCoding);
26     provider.document.transact(() => {
27         YjsEditor.removeStoredPosition(editor,
  ↪ `${updatedCoding.id}.start`);
28         YjsEditor.removeStoredPosition(editor,
  ↪ `${updatedCoding.id}.end`);
29     });
30 }
31 });
32 };
```

---

**Listing 5.2:** Preview Update

## PDFs

Im Vergleich zur Anbindung der Textdokumente waren die notwendigen Änderungen zur Anbindung der PDFs deutlich geringer. Bedingt durch die statische Natur von PDFs, mussten hier nur die Erstellung, das Updaten und Löschen von Kodierungen infolge einer direkten Interaktion des Nutzers mit der `CodesystemToolBar` beachtet werden.

Hierbei konnte auf die bereits für die Textdokumente entwickelte Logik zurückgegriffen werden, wobei das jeweilige Update der `stored positions` ausgelassen werden konnte. Dadurch werden erstellte Text- und Bereichskodierungen in Echtzeit synchronisiert.

## Aktive Nutzer im Projekt

Auch die Funktion des RTCS, aktive Nutzer im Projekt (*Abbildung 5.1*) und deren aktuell bearbeitetes Dokument (*Abbildung 5.2*) anzuzeigen, musste in den CES übertragen werden.



Abbildung 5.1: Anzeige aktiver Projekt-Nutzer in der Editor-ToolBar



Abbildung 5.2: Anzeige aktiver Nutzer in der Dokumentübersicht

Hierfür wurde die Awareness-Funktion<sup>2</sup> von `Yjs` verwendet. Über diese werden in 15-sekündigen Intervallen die notwendigen Nutzerinformationen an den CES gesendet, wie in *Listing 5.3* dargestellt. Dieses Intervall wurde gewählt, da `Yjs` einen Nutzer als inaktiv klassifiziert und damit die Nutzerinformation aus dem lokalen

<sup>2</sup> <https://docs.yjs.dev/getting-started/adding-awareness>

Cache entfernt, wenn 30 Sekunden keine Awareness-Nachricht von einem Nutzer eintrifft. Das 15-sekündige Intervall sorgt außerdem dafür, dass neue Nutzer zeitnah über alle im Projekt aktiven Mitarbeiter informiert werden, wobei der Informationsversand nicht zu unnötigem Overhead führt.

Diese Daten werden über den CS-Provider propagiert, da ein CS immer genau einem Projekt zugeordnet wird und entsprechend genau ein Yjs-Kanal pro Projekt angelegt wird. So stehen diese Informationen unabhängig vom aktuell verwendeten Dokument zur Verfügung.

Neben den Informationen, die an dieser Stelle bereits über den RTCS synchronisiert wurden, wird hier zusätzlich noch die Id des aktuell geöffneten Dokuments übermittelt. Diese Information wurde im Kontext des RTCS über eine eigene Nachricht propagiert. Da sie jedoch genauso wie die allgemeinen Nutzerinformationen für alle Nutzer im Projekt interessant ist, kann sie an dieser Stelle ohne großen Mehraufwand mitgesendet werden.

---

```
"user": {
  userId: number, // Nutzer-Id als Identifikator
  email: string,
  name: string, // Vor- + Nachname
  picSrc: string, // URL zu Profilbild
  document: string // Id des geöffneten Dokuments
}
```

---

**Listing 5.3:** Repräsentation der Awareness-Daten-Struktur für Nutzer im Projekt

### Aktive Nutzer im aktuellen Textdokument

Neben der Information über aktive Nutzer im allgemeinen, sollten auch die Positionen der Cursor bzw. die gerade ausgewählten Textpassagen der anderen Nutzer im aktuellen Textdokument angezeigt werden.

Hierfür wurde auf das `Cursor Plugin`<sup>3</sup> von `slate-yjs` in Verbindung mit dem `Cursor Overlay`<sup>4</sup> zurückgegriffen.

Alternativ wäre auch die Verwendung der `Cursor Decorations`<sup>5</sup> möglich gewesen, wobei diese laut `slate-yjs` Dokumentation (2022) im Vergleich zu den verwendeten Overlays einige Nachteile haben:

- Sie werden als Teil des `Slate`-Editors angelegt und können deshalb durch Änderungen der Markierung eines Nutzers zu anderen Umbrüchen und Neu-Rendering führen

---

<sup>3</sup> <https://docs.slate-yjs.dev/api/slate-yjs-core/cursor-plugin>

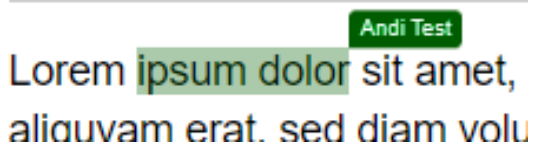
<sup>4</sup> <https://docs.slate-yjs.dev/api/slate-yjs-react/cursor-overlay>

<sup>5</sup> <https://docs.slate-yjs.dev/api/slate-yjs-react/cursor-decorations>

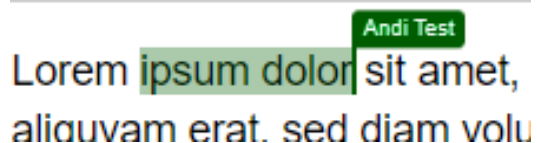
- Sie können die Autokorrektur des Browser beeinflussen.
- Sie können zu schlechterer Performance führen, da bei Änderungen Teile des Editors neu gerendert werden müssen.

Aufgrund dieser Nachteile wurde sich hier für die **Cursor Overlays** entschieden. Dabei wurden drei verschiedene Anzeige-Versionen für vier Szenarien angelegt:

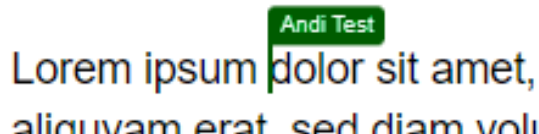
- Selektion im Kodierungseditor: Zeigt die Selektion des Nutzers, blendet aber das Caret aus, da der Nutzer den Text nicht bearbeiten kann (*Abbildung 5.3*).
- Cursor im Kodierungseditor: Wird nicht angezeigt, da der Nutzer den Text nicht bearbeiten kann.
- Selektion im Texteditor: Zeigt die Selektion des Nutzers und das Caret (*Abbildung 5.4*).
- Cursor im Texteditor: Zeigt das Caret des Nutzers (*Abbildung 5.5*).



**Abbildung 5.3:** Anzeige einer Selektion im Kodierungseditor (ohne Caret)



**Abbildung 5.4:** Anzeige einer Selektion im Texteditor (mit Caret)



**Abbildung 5.5:** Anzeige der Position eines Nutzers im Texteditor

Um diese Anzeige zu ermöglichen, wurde der Yjs-Provider des Texteditors um den Versand eines weiteren Awareness-Feldes erweitert, der den aktuell gewählten Editor des Nutzers beinhaltet (*Listing 5.4*).

```
"editor": {
  userId: number, // Nutzer-Id als Identifikator
  currentEditorMode: string // 'coding' oder 'text'
}
```

**Listing 5.4:** Repräsentation der Awareness-Daten-Struktur für den Editorstatus eines Nutzers im Textdokument

Da der PDF-Editor keine eigene `Yjs`-Instanz besitzt und nicht auf `Slate` basiert, wurde hier zunächst auf die Einführung der Selektionsanzeigen verzichtet. Eine Cursoranzeige kommt bei PDFs als nicht-editierbarer Dokumenttyp nicht in Frage, da dieser eine Bearbeitungsmöglichkeit suggerieren würde, welche nicht vorhanden ist.

### 5.1.2 Anbindung weiterer Features an den CES

Während der Umsetzung dieser Arbeit wurden weitere Features in `QDAcity` integriert, wobei die Benutzung im kollaborativen Umfeld nicht oder nur unter Verwendung des `RTCS` vorgesehen wurde. Diese Features mussten zusätzlich im Rahmen dieser Arbeit auf die Nutzung von `Yjs` umgestellt werden. Die Umstellung dieser Features wird hier erläutert.

#### Empfehlungen

Als erstes neues Feature wurden die von Schöpf (2023) eingeführten Empfehlungen in den `CES` integriert.

Dieses bietet den Nutzern die Möglichkeit, Empfehlungen für die Erstellung, das Anpassen oder Löschen von Codes einzubringen. Folgend können andere Nutzer über die eingereichten Empfehlungen abstimmen. Mitarbeiter mit den Rollen „Organizer“ oder „Owner“ können Empfehlungen abschließend anwenden oder ablehnen.

Da insbesondere das Voting-System komplexere Logik beinhaltet und hier die Manipulation der Votes verhindert werden soll, wurde bei der Anbindung dieses Features auf einen hybriden Ansatz gesetzt. Dabei wird das Java-Backend weiterhin zum Verarbeiten und Persistieren aller Änderungen angesprochen und die resultierende Antwort wird lediglich über `Yjs` propagiert.

Dafür wird das `CS-yDoc` um eine weitere `yMap` ergänzt, in welcher die Empfehlungsobjekte propagiert werden. Beim Erstellen des `CS-yDoc` wird diese Map im `CES` mit den aktuellen Daten aus dem Datastore befüllt. Im Laufe der Nutzung des `CS` werden dann alle Änderungen der einzelnen Clients in die Map übernommen und in Echtzeit propagiert. Dafür konnten in der Frontend-`CS`-Komponente die bereits vorhandenen Callbacks um den Aufruf der Aktualisierungsfunktion des `CS-Providers` ergänzt werden.

#### Benutzerdefinierte-Editoren

Neben den Empfehlungen wurden auch die neu eingeführten „Benutzerdefinierten-Editoren“ angepasst. Diese ermöglichen eine freie Auswahl der zur Verfügung stehenden Editorkomponenten.

Dabei wurden dieselben Komponenten verwendet, die auch in den klassischen Editoransichten genutzt werden. Daher musste hier nur entsprechend die Umstellung der übergebenen Eigenschaften auf die im kollaborativen Umfeld verwendeten Typen durchgeführt werden.

## 5.2 Migration aller kollaborativer Daten

Neben der Anbindung des CES an die Komponenten des Frontends musste auch ein Migrationsmechanismus geschaffen werden, um bestehende Projekte ohne Datenverlust auf die neuen Datenstrukturen zu migrieren.

### 5.2.1 Migration der Kodierungen

Die zugrundeliegende Problematik bei der Migration der Kodierungen wurde bereits in 4.3.3 beschrieben.

Im Rahmen der Umsetzung stellte sich heraus, dass die angenommene Abweichung durch die fehlende Browser-API in NodeJS tatsächlich die Erstellung der Kodierungen im CES verhinderte.

Zwar konnte die richtige Position im Dokument als `Slate Point`<sup>6</sup> und entsprechend die dem Coding entsprechende `Slate Range`<sup>7</sup> abgeleitet werden, jedoch konnte diese nicht als `stored positions` im Dokument hinterlegt werden.

Um nicht das Dokument im neuen und alten Format zur Initialisierung ans Frontend senden zu müssen, sondern den Datenverkehr auf ein notwendiges Minimum zu beschränken, wurde der Migrationsprozess zweigeteilt:

Der CES berechnet im Rahmen der initialen Migration eines TEXT-Dokuments die `Slate Range` für jede Kodierung und legt diese als zusätzliche Map unter dem Key `migrationCodings` im `yDoc` des `COLLABORATIVETEXT`-Dokuments ab.

Das Frontend prüft beim Laden eines `COLLABORATIVETEXT`-Dokuments, ob die `migrationCodings`-Map Elemente enthält. Ist dies der Fall, so legt es die benötigten `stored positions` am Start- und Endpunkt jeder Kodierung an und entfernt diese daraufhin aus der Map.

In der Kodierungs-Map des `CS-yDocs` ist die Definition dieser Kodierung bereits vorhanden, daher müssen hier nur die `stored positions` angelegt werden.

## 5.3 Anpassungen des CES

Im Rahmen der zuvor beschriebenen Änderungen mussten auch einige Anpassungen am CES vorgenommen werden, welche folgend beschrieben werden.

---

<sup>6</sup> <https://docs.slatejs.org/v/v0.47/slate-core/point>

<sup>7</sup> <https://docs.slatejs.org/v/v0.47/slate-core/range>

### 5.3.1 Prüfung der Autorisierung eines Nutzers

Da Yjs alle durchgeführten Änderungen zwischen den Clients synchronisiert und abschließend im CES persistiert, musste auch die rollenbasierte Zugriffskontrolle (engl. *Role-Based Access Control*) (RBAC) von QDAcity berücksichtigt werden. Diese vergibt einem Nutzer Berechtigungen auf Projektebene, die in verschiedene Berechtigungsgruppen unterteilt sind.

In QDAcity dürfen Projektmitglieder mit der Rolle „Viewer“ keine Änderungen an den Dokumenten und dem CS vornehmen. Damit diese Anforderung erfüllt bleibt, wurde die Prüfung der Autorisierung eines Nutzers im CES erweitert. Ursprünglich wurde hier beim Aufbau der Websocket-Verbindung für ein yDoc nur geprüft, ob der Nutzer (authentisiert via JWT) Zugriff auf das angeforderte Dokument hat. Dafür wurden bei Dokumenten die Metainformationen des Dokuments vom Backend abgefragt. Zur Verifizierung des Zugriffs auf das CS wurde dieses komplett geladen. Wurde diese Anfrage vom Backend verweigert, so wurde der Nutzer als „nicht berechtigt“ abgelehnt und die Websocket-Verbindung wurde nicht aufgebaut. Dieses Vorgehen ist enorm ineffizient, da die zur Prüfung der Autorisierung geladenen Daten nach Verbindungsaufbau nochmals geladen werden müssen, um das yDoc initial zu befüllen. Zudem kann so keine Unterscheidung nach der Nutzergruppe, die ein Anwender innerhalb des betrachteten Projektes hat, durchgeführt werden, da diese Information weder im JWT noch in der Antwort des Backends vorhanden ist.

Um die RBAC-Berechtigungsprüfung auch im CES durchführen zu können, wurde das Backend um einen neuen Endpoint erweitert. Dieser prüft nach Übermittlung der Projekt-Id, des Projekt-Typs, eines JWT und einer Berechtigungsgruppe, ob ein Nutzer die entsprechende Berechtigung besitzt. Wird keine Berechtigungsgruppe angegeben, so wird die Mitgliedschaft im angegebenen Projekt geprüft. Das Ergebnis wird mittels HTTP-Statuscode zurückgeliefert, wobei ein Status von 204 *No Content* die Berechtigung eines Nutzers signalisiert, während der Status 403 *Forbidden* bei fehlender Genehmigung zurückgegeben wird.

So kann bei Aufruf eines Dokuments direkt die entsprechende Berechtigung zum Zugriff auf, sowie zum Modifizieren des Projekts (*EDITOR\_CUD*, abgeleitet von *CREATE*, *UPDATE*, *DELETE*) geprüft werden. Da diese Anfragen, mit Ausnahme des Payloads zur Definition der Anfrage, keine Daten übertragen, sind sie gerade bei großen Projekten deutlich effizienter als das vorherige Vorgehen.

Ist ein Nutzer Mitglied im angefragten Projekt, so wird die Websocket-Verbindung initialisiert. Fehlt ihm die Berechtigung zum Bearbeiten, so wird die Verbindung auf *readonly* gesetzt, wodurch lokale Änderungen nicht von Yjs propagiert werden.

Darüber hinaus wird auch vor jeder eingehenden Nachricht geprüft, ob das JWT des Nutzers noch gültig ist. Ist das Token abgelaufen, so wird die Websocket-Verbindung geschlossen und dem Client mittels des Status-Codes 4440 mitgeteilt,



dass der Grund für die Schließung sein abgelaufenes Token ist. Das Frontend kann daraufhin die Verbindung mit einem aktualisierten Token wieder aufbauen.

### 5.3.2 Einführung einer Systemidentität

Neben der Prüfung der Autorisierung eines Nutzers sollte der CES zudem auch eine eigene Identität bekommen, um Änderungen eigenständig am Backend persistieren zu können. Der bisherige Mechanismus verwendete hier das JWT, mit welchem sich der erste Nutzer beim initialen Verbindungsaufbau mit dem CES authentifiziert hat. Dieses hat jedoch nur eine Gültigkeit von 40 Minuten, wodurch das Backend alle Anfragen mit diesem Token nach Ablauf der Zeitspanne ablehnt. Das hatte in der ursprünglichen Implementierung zur Folge, dass Änderungen, die kurz vor Ablauf der Gültigkeit gemacht wurden, nicht mehr am Backend persistiert werden konnten. Außerdem konnte ein yDoc, das zuerst von einem Nutzer mit „Viewer“-Berechtigung geöffnet wurde und folgend zusätzlich von einem Nutzer mit Schreibrechten angefordert wurde, die Änderungen des berechtigten Nutzers nicht persistieren. Grund dafür ist, dass der CES die Speicherungsanfrage mit dem JWT des ersten verbundenen Nutzers an das Backend sendet. Dieses lehnt in dem beschriebenen Fall die Änderungen korrekterweise ab, da der Nutzer keine Schreibberechtigung im Projekt inne hat.

Um das zu verhindern, wurde ein weiteres Google Cloud Dienstkonto angelegt, für das der CES ein eigenes OAuth-Token von Google anfordert. Hierfür wird ein JWT im CES generiert, welches mit einem beim Erzeugen des Dienstkontos generierten Private Key signiert wird. Google prüft dieses signierte JWT und gibt dann ein OAuth-JWT zurück, mit welchem sich der CES am Backend authentifizieren kann. Im Backend konnte hier die bestehende Logik zur Authentifizierung von Google OAuth-Token verwendet werden. Hier musste nur beim Start des Servers eine Methode eingefügt werden, die einen entsprechenden Nutzer in der Datenbank anlegt. Dies wurde in Form eines `Servlets` umgesetzt, das beim Starten des Servers prüft, ob es einen Nutzer mit der OAuth-Id des CES-Dienstkontos in der Datenbank gibt. Ist dies nicht der Fall, so wird der entsprechende Nutzer als Administrator angelegt.

Bei einer Anfrage wird, nach der Prüfung des übergebenen Tokens auf Validität, dieser QDacity-Nutzer aus der Datenbank geladen. Durch die Administratorberechtigung ist er berechtigt, alle angeforderten Änderungen durchzuführen. Durch diesen Mechanismus ist der CES zukünftig in der Lage, unabhängig von der verbleibenden Gültigkeit eines Nutzertokens, Änderungen am Backend zu persistieren. Durch die zuvor beschriebenen Anpassungen zur Prüfung der Autorisierung eines Nutzers innerhalb des CES, kann es dabei auch nicht zur Persistierung von unautorisierten Änderungen kommen.



# 6 Evaluation

In diesem Kapitel soll die Erfüllung der in *Kapitel 3* gestellten Anforderungen an die im Rahmen dieser Arbeit umgesetzten Implementierungen bewertet werden. Dabei wird zunächst die Umsetzung der FA geprüft, bevor auf die NFA eingegangen wird.

## 6.1 Funktionale Anforderungen

**FA-1:** Yjs sollte alle Daten, die für die Nutzer von Relevanz sind, in Echtzeit zwischen den Nutzern synchronisieren.

**FA-1.1:** Sobald Änderungen am Codesystem wie Erstellen, Umbenennen, Änderungen an der hierarchischen Struktur oder Löschen von Codes vorgenommen werden, muss Yjs diese Änderungen synchronisieren.

**FA-1.2:** Sobald Änderungen an den mit dem Codesystem verknüpften Entitäten `Memo`, `CodeRelation` und `CodebookEntry` vorgenommen werden, muss Yjs diese Änderungen synchronisieren.

**FA-1.3:** Sobald Änderungen am Dokumentinhalt wie Kodierungen oder Textanpassungen bei Textdokumenten vorgenommen werden, muss Yjs diese synchronisieren.

**FA-1.4:** Sobald Änderungen an Dokument-Metadaten, also das Erstellen, Umbenennen oder Löschen, vorgenommen werden, sollte Yjs andere Clients über diese Änderungen informieren.

Die Frontend-Einbindung von Yjs ermöglicht die sofortige Synchronisation aller Änderungen an Codes (einschließlich `Memo`, `CodeRelation` und `CodebookEntry`). Die Einbindung in den Texteditor ermöglicht außerdem auch, alle darin vorgenommenen Änderungen zu synchronisieren. Durch die Verwendung von Yjs wird zudem verhindert, dass sich gleichzeitige Änderungen im gleichen Text-

abschnitt gegenseitig überschreiben (Dürsch, 2023). Meldungen über Änderungen an Dokument-Metadaten werden nach der Persistierung am Backend über das CS-yDoc an alle anderen Clients propagiert. Somit ist die Anforderung als erfüllt zu betrachten.

### Die Anforderung FA-1 wurde erfüllt.

**FA-2:** Yjs sollte Awareness-Informationen über den Nutzer an andere Clients senden.

**FA-2.1:** Falls ein Nutzer im Projekt aktiv ist, muss diese Information in der Nutzeranzeige angezeigt werden.

**FA-2.2:** Sobald ein Nutzer ein Dokument geöffnet hat, sollte dies in der Dokumentenanzeige angezeigt werden.

**FA-2.3:** Sobald der Texteditor geöffnet ist, muss der Editor die Textselektierung eines Nutzers anzeigen.

**FA-2.4:** Sobald ein Nutzer im Texteditiermodus ist, muss der Editor den Cursor des anderen Nutzers anzeigen.

**FA-2.5:** Sobald ein Bereich eines PDFs ausgewählt ist, sollte der Editor die Selektion des anderen Nutzers anzeigen.

QDAcity wurde entsprechend angepasst, sodass alle aktiven Benutzer (einschließlich des aktuellen Benutzers, so in einem anderen Browsertab aktiv) im Projekt angezeigt werden. Dabei wurde die Funktionalität zur Anzeige aller Nutzer in den Dokumenten des Projekts beibehalten und zudem die Anzeige der aktuellen Position bzw. Selektierung in Textdokumenten hinzugefügt.

Für PDFs wurde die Bereichsanzeige wie in 5.1.1 begründet nicht implementiert. Im Rahmen dieser Arbeit wurde diese Funktion als vergleichsweise weniger relevant betrachtet und daher für eine zukünftige Implementierung umterminiert.

### Die Anforderung FA-2 wurde größtenteils erfüllt.

**FA-3:** Der CES muss fähig sein, nur berechtigten Servern von QDAcity die angeforderten Daten zu liefern und nicht autorisierte Anfragen abzulehnen.

**FA-3.1:** Zur Authentifizierung einer Anfrage sollen keine Autorisierungsrundläufe notwendig sein.

Die konzipierte Authentifikation mittels mTLS erlaubt eine direkte Autorisierung einer Anfrage durch Prüfung des übermittelten Zertifikats. Leider konnte die

Implementierung aus Zeitgründen im Rahmen dieser Arbeit nicht mehr erfolgen, wodurch diese Anforderung als nicht erfüllt betrachtet werden muss.

**Die Anforderung FA-3 wurde nicht erfüllt.**

## 6.2 Nicht-funktionale Anforderungen

**NFA-1:** Der CES muss so gestaltet sein, dass er innerhalb der von Google Cloud Run zur Verfügung gestellten Umgebung ausgespielt und betrieben werden kann.

**NFA-1.1:** Google Cloud Run bietet ein Maximum von 32 Gigabyte Arbeitsspeicher pro Instanz.<sup>1</sup>

**NFA-1.2:** Websocket-Verbindungen unter Google Cloud Run müssen innerhalb einer Stunde abgeschlossen sein.<sup>1</sup>

Nach ersten Tests benötigt der CES zum Starten eine Mindestmenge von 80 MiB RAM, wobei er ohne verbundene Nutzer nur 50 MiB RAM verwendet. Dabei erhöhen die geladenen Dokumente den benötigten Arbeitsspeicher weiter, wobei das Limit von 32 GB wohl nur unter extremer Last erreicht werden könnte. Auf einen Lasttest wurde aufgrund des Zeitrahmens verzichtet, da Dürsch (2023) bei 15-20 seitigen Dokumenten bereits RAM-Auslastung im einstelligen MiB-Bereich verzeichnen konnte. Somit können mit 100 MiB freiem RAM potentiell 50 Nutzer in jeweils eigenen Dokumenten von einer einzigen CES-Instanz bedient werden. Dies würde der aktuellen Konfiguration des RTCS entsprechen, welcher auf 160 MiB begrenzt ist. Das Limit von einstündigen Verbindungen ist für den CES nicht relevant, da das Frontend bei Abbruch der Verbindung sofort eine neue Verbindung initiieren kann. Dies kann ohne für den Nutzer ersichtliche Seiteneffekte erfolgen. NFA-1 wurde folglich erfüllt.

**Die Anforderung NFA-1 wurde erfüllt.**

**NFA-2:** Die GitLab Deployment-Pipeline muss so gestaltet sein, dass der CES bei jedem Deployment von QDAcity mit deployed wird.

**NFA-2.1:** Die GitLab Deployment-Pipeline muss so gestaltet sein, dass die Environment-Variablen des CES auf die aktuelle Backend-Url verweisen.

Das Deployment ist entsprechend der Anforderungen vorbereitet, um den CES mit jedem Release auf den jeweiligen Server zu deployen. Dabei werden alle notwen-

---

<sup>1</sup> <https://cloud.google.com/run/quotas>

digen Umgebungsvariablen gesetzt. Aufgrund letzter fehlender Features findet aktuell jedoch noch kein Einsatz des CES in der Produktionsumgebung statt.

### Die Anforderung NFA-2 wurde erfüllt.

**NFA-3:** Der CES muss so gestaltet sein, dass alle bestehenden Features erhalten bleiben und in Zukunft leicht neue Features implementiert werden können.

**NFA-3.1:** Der CES muss so gestaltet sein, dass jegliche bestehende Funktionalität erhalten bleibt.

**NFA-3.2:** Der CES muss so gestaltet sein, dass zukünftig weitere Dokumenttypen synchronisiert werden können.

**NFA-3.3:** Der CES muss so gestaltet sein, dass er in Zukunft Offline-Bearbeitung unterstützen kann.

**NFA-3.4:** Der CES muss so gestaltet sein, dass in Zukunft auch weitere Features leicht unterstützt werden können.

Die Erweiterungen des CES im Rahmen dieser Arbeit dienten dazu, alle bisherigen Funktionen des RTCS abzubilden. So werden jetzt alle kollaborativen Daten, sowie Awareness-Informationen synchronisiert. Außerdem wurden alle Änderungen so vorgenommen, dass mit jedem abgeschlossenen Feature alle anderen Funktionalitäten voll kompatibel mit den Änderungen sind. So werden aktuell die Daten des CS weiterhin im Datastore synchron gehalten, obwohl die primäre Speicherung innerhalb des CS-yDocs stattfindet, um weiterhin Auswertungen im Backend zu ermöglichen.

Des Weiteren ist der CES für die einfache Erweiterung um weitere Dokumenttypen konzipiert. Dies wurde zum Teil schon bei der Umsetzung der Unterstützung für PDF-Dateien bewiesen, wobei hier nur die Kodierungen und nicht das eigentliche Dokument über den CES synchronisiert werden.

Yjs bietet Unterstützung für Offline-Bearbeitungen mit folgender Synchronisation zwischen Clients. Somit ist der CES derart gestaltet, dass Erweiterungen in der Zukunft leicht unterstützt werden können.

Abschließend ist der CES auch so gestaltet, dass neue Features leicht unterstützt werden können. Dies wurde im Rahmen dieser Arbeit bereits an den Erweiterungen um die Awareness-Informationen und die Recommendations gezeigt. Somit ist die Anforderung als erfüllt zu betrachten.

### Die Anforderung NFA-3 wurde erfüllt.

## 7 Hindernisse bei der Umsetzung

In diesem Kapitel werden kurz die Probleme beschrieben, welche die Weiterentwicklung von `QDAcity` im Rahmen dieser Arbeit erheblich behindert haben. So wurde die Umsetzung dieser Arbeit durch fehlende Dokumentation und daraus resultierendes fehlendes Verständnis der Interaktionen einiger Komponenten, sowie die teilweise schlechte Qualität der Codebasis im Frontend von `QDAcity` wiederholt verzögert. Es kann nur eindringlich dazu geraten werden, vor zusätzlichen Erweiterungen von `QDAcity` zunächst die bestehende Codebasis zu überarbeiten und ausführlicher zu dokumentieren. Eine umfangreiche und präzise Dokumentation ist von essentieller Bedeutung, um eine nachhaltige und effiziente Weiterentwicklung von `QDAcity` zu gewährleisten. Der aktuelle Zustand der Codebasis weist zwar teilweise kommentierte Methoden auf, jedoch fehlt eine umfassende Übersicht über die verschiedenen Komponenten von `QDAcity`. Dies führt zu einer erheblichen Einschränkung der Fähigkeit, schnell und präzise die Zusammenhänge innerhalb des Systems zu erfassen.

Im Idealfall kann die durch diese Arbeit gebotene Dokumentation hierfür als Ausgangspunkt genommen werden, um zumindest die Datenstruktur, sowie das kollaborative Editieren und seine Bestandteile zu erklären.

Auf diese Weise kann in Zukunft dazu beigetragen werden, frühzeitige Fehlannahmen zu vermeiden. Außerdem wird das Risiko reduziert, die komplexen Abhängigkeiten im Projekt erst nach begonnener Implementierung zu erkennen. Indem alle wesentlichen Informationen über das System klar und transparent dargelegt werden, können potenzielle Fallstricke und Herausforderungen frühzeitig erkannt und vermieden werden.

## 7. Hindernisse bei der Umsetzung

---



## 8 Fazit

Abschließend soll in diesem Kapitel ein Résumé über diese Arbeit gezogen werden. Hierfür werden die Ergebnisse der Arbeit kurz zusammengefasst, bevor ein Ausblick auf mögliche zukünftige Weiterentwicklungen gegeben wird.

Im Rahmen dieser Arbeit konnte das kollaborative Editieren innerhalb von `QDacity` erheblich weiterentwickelt werden. So werden nun alle aktuell implementierten Features des Frontends unterstützt. Außerdem wird in Echtzeit angezeigt, an welchem Dokument andere Nutzer aktuell arbeiten und bei Textdokumenten wird sogar die aktuelle Selektion hervorgehoben. Durch diese Features wird die Kollaboration innerhalb von `QDacity` auf den aktuellen technischen Stand anderer Kollaborationsprogramme wie Office365 oder Google Docs angehoben.

Um diese Transformation abzuschließen, fehlt jedoch noch die Generierung von ICA-Berichten, da dem Backend dafür aktuell nicht das richtige Format der Textdokumente zur Verfügung steht.

Aus diesem Grund sollte als nächster Schritt die im Rahmen dieser Arbeit bereits skizzierte REST-API zum Abruf aller für die Berichterstellung notwendiger Daten implementiert werden. Erst nach dieser Umsetzung kann das kollaborative Editieren unter Verwendung des `CES` die aktuelle Implementierung des `RTCS` ersetzen.

Weiteres Verbesserungspotential findet sich auch im Audiotranskript-Editor, welcher aktuell nicht auf `Slate` basiert und daher auch nicht über den `CES` synchronisiert werden kann. Hier würde eine auf `Slate` basierende Implementierung die zukünftige Entwicklung stark begünstigen, da dann statt drei verschiedener Editoren nur noch zwei Editoren (`Slate`, `PDFs`) aktualisiert und weiterentwickelt werden müssten.

Eine dritte wichtige Weiterentwicklung wäre zudem die Umstellung der ICA-Berichterstellung auf einen `NodeJS`-basierten Server, wahlweise entweder als weitere Funktionalität innerhalb des `CES` oder in Form eines eigenständigen Services. Diese Erweiterung ersetzt zwar die REST-API zur Datenabfrage, welche jedoch vor allem eine schnelle Lösung dar stellen soll, die Neuerungen des `CES` für die Nutzer zugänglich zu machen. Durch die Umstellung auf das `yDoc`-Format kann die indirekte Referenzierung via `codingkeys` ersetzt werden und der Aufwand für

## 8. Fazit

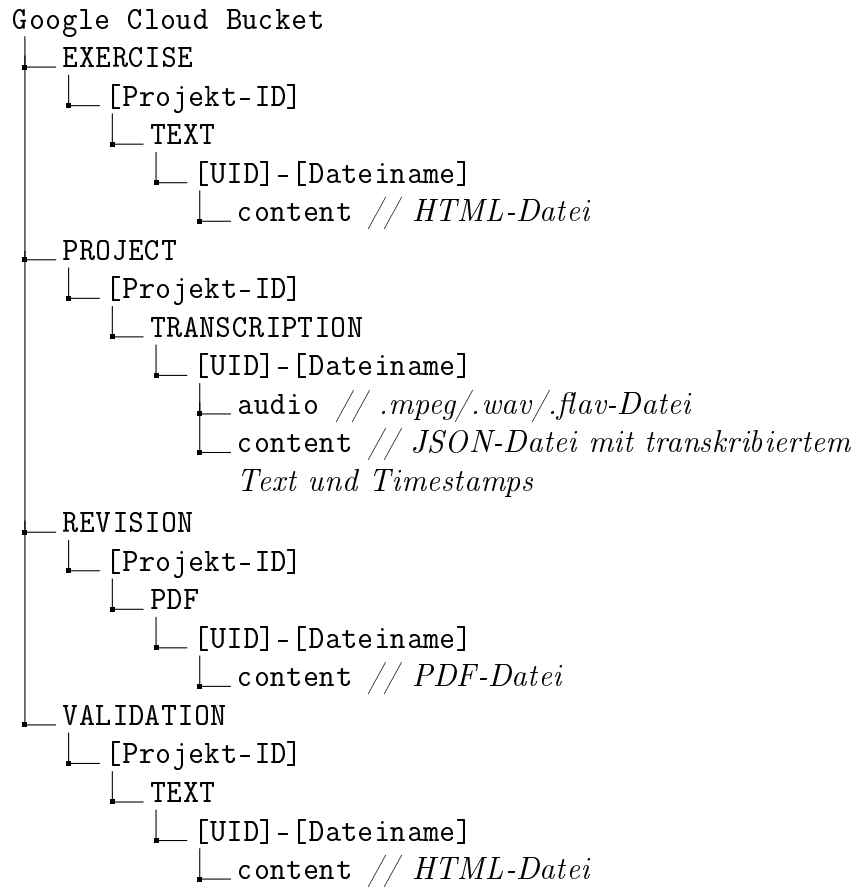
---

die Umwandlung der Dokumente und des CS wird eingespart. Die Speicherstruktur der `yDocs` würde zudem theoretisch auch weitere Berichte ermöglichen, da damit auch der Verlauf der Übereinstimmung über die Zeit oder andere historische Daten direkt auf dem Dokument ausgewertet werden können.

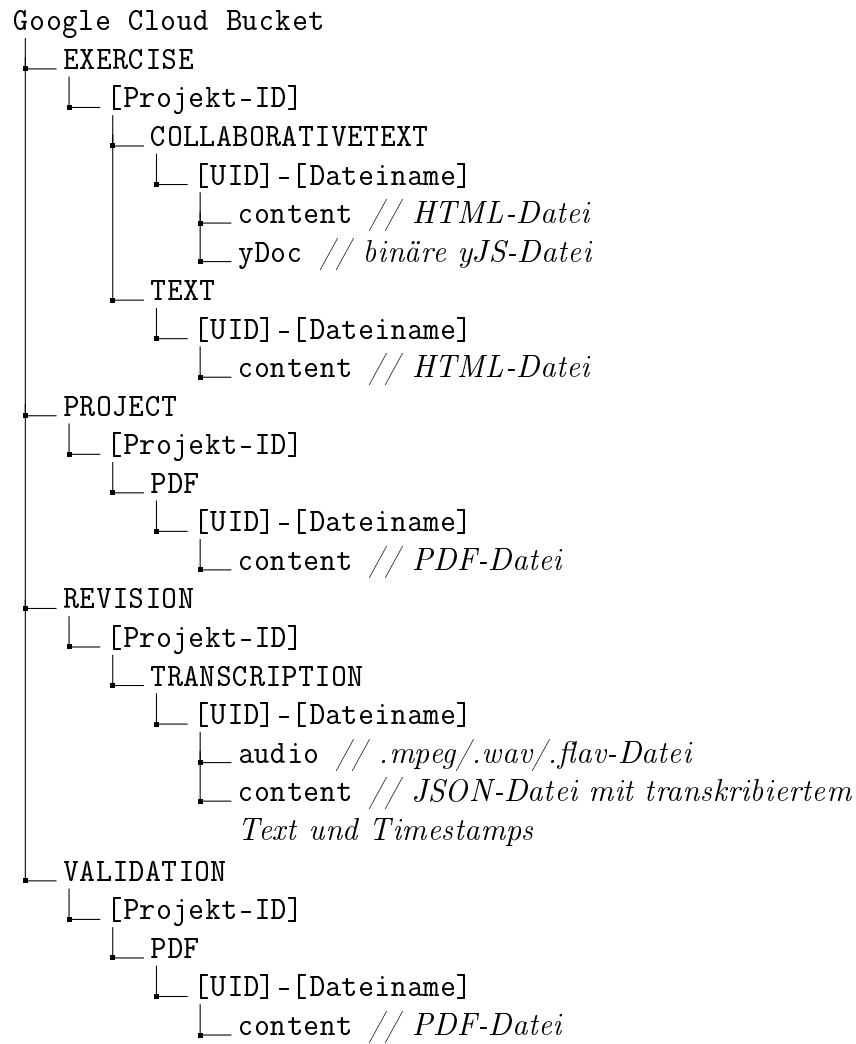
# Anhänge



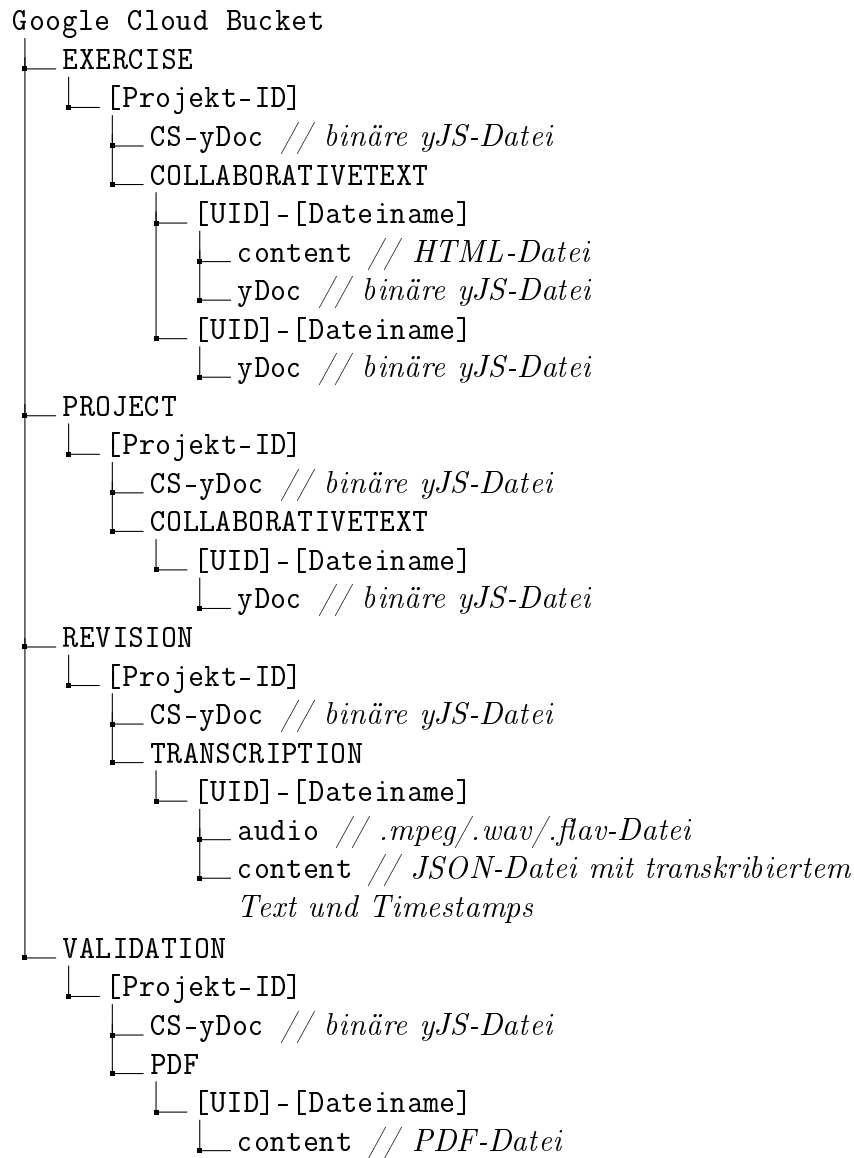
## A Datenstruktur



**Abbildung 1:** Vollständige Bucket-Struktur eines QDAcity-Buckets vor Einführung des kollaborativen Editierens



**Abbildung 2:** Vollständige Bucket-Struktur eines QDAcity-Buckets nach Einführung des kollaborativen Editierens



**Abbildung 3:** Geplante vollständige Bucket-Struktur eines QDAcity-Buckets





# Literaturverzeichnis

- Anavatti, V. (2022). *Intercoder Evaluation Metrics in QDAcity* [Masterarbeit]. Friedrich-Alexander-Universität Erlangen-Nürnberg.
- auth0 by Okta. (n. d.). *Libraries for Token Signing/Verification*. <https://jwt.io/libraries>. Aufgerufen am: 8. Juni 2023.
- Cloudflare. (n. d.). *Was ist mutual TLS (mTLS)?* <https://www.cloudflare.com/de-de/learning/access-management/what-is-mutual-tls/>. Aufgerufen am: 10. Juni 2023.
- Dürsch, M. (2023). *Scaling Real-time Collaborative Editing in a Cloud-based Web App* [Masterarbeit]. Friedrich-Alexander-Universität Erlangen-Nürnberg.
- Elektronik Kompendium. (n. d.). *TLS - Transport Layer Security*. <https://www.elektronik-kompendium.de/sites/net/1706131.htm>. Aufgerufen am: 11. Juni 2023.
- Graue, C. (2015). Qualitative Data Analysis. *International Journal of Sales, Retailing and Marketing*, 4(9), 5–14.
- Jones, M. B., Campbell, B., & Mortimore, C. (2015). JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants. <https://doi.org/10.17487/RFC7523>
- Jones, M. B., & Hardt, D. (2012). The OAuth 2.0 Authorization Framework: Bearer Token Usage. <https://doi.org/10.17487/RFC6750>
- Lesavre, L., Varin, P., & Yaga, D. (2021). *Blockchain Networks: Token Design and Management Overview* (Techn. Ber.). National Institute of Standards; Technology. <https://doi.org/10.6028/nist.ir.8301>
- Rupp, C., & Joppich, R. (2014). Anforderungsschablonen – der MASTER-Plan für gute Anforderungen. In *Requirements-Engineering und -Management* (S. 215–246). <https://doi.org/10.3139/9783446443136.010>
- Schöpf, D. (2023). *Recommendation System for Qualitative Data Analysis* [Masterarbeit]. Friedrich-Alexander-Universität Erlangen-Nürnberg.
- slate-yjs Dokumentation. (2022, 1. Oktober). *Cursor Decorations*. <https://docs.slate-yjs.dev/api/slate-yjs-react/cursor-decorations>. Aufgerufen am: 24. August 2023.

- Springer. (n. d.). *Lehrbuch Psychologie: Qualitative Datenanalyse*. <https://lehrbuch-psychologie.springer.com/glossar/qualitative-datenanalyse>. Aufgerufen am: 9. September 2023.
- VaibhavBilotia, pall58183, saurabh1990aror, rkbhola5 & sashwatdesai. (2022, 18. Oktober). *This is exactly why we still use the OSI model when we have TCP/IP Model*. <https://www.geeksforgeeks.org/this-is-exactly-why-we-still-use-the-osi-model-when-we-have-tcp-ip-model/>. Aufgerufen am: 12. Juni 2023.