Bachelor's Thesis

Providing a distributed file storage for German Schul-Cloud

Erstellung einer verteilten Dateiverwaltung für die deutsche Schul-Cloud

by

Niklas Kiefer

Potsdam, June 2017

Supervisor

Prof. Dr. Christoph Meinel,
Jan Renz

Internet-Technologies and Systems Group

Disclaimer

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Potsdam, 16. Juni 2017

(Niklas Kiefer)

Kurzfassung

Einer der wichtigsten Vorhaben der deutschen Schul-Cloud ist es, verschiedenartige Unterrichtsmaterialien für jedermann zugänglich und auf mehreren Geräten verfügbar zu machen. Es soll möglich sein, Dateien kollaborativ zu teilen und im Unterricht ohne zusätzliche Software einzusetzen.

Das Ziel der vorliegenden Bachelorarbeit war es, eine verteilte und damit flexible Dateiverwaltung für das Schul-Cloud zu entwerfen. Es wurden bereits bewährte File Storage Anbieter evaluiert, Studien im Bereich digitalen Lernens ausgewertet und daraus ein dynamisches Konzept für eine Dateiablage entwickelt. Verschiedene Implementierungen im derzeitigen Schul-Cloud Server und Client wurden aufgezeigt und mit anderen Lösungen verglichen. Zudem wurde eine Reihe von Anschlussmöglichkeiten für die Dateiverwaltung aufgezählt.

Abstract

One of the major purposes of German Schul-Cloud is to make various teaching materials accessible to everyone and to make them available on several devices. It should be possible to share files collaboratively and use them in lessons without additional software.

The goal of the present Bachelor thesis was to design a distributed and flexible file storage for the Schul-Cloud. Existing file storage providers have been evaluated, studies in the area of digital learning have been checked and a dynamic concept for a file storage has been developed. Several implementations in the current Schul-Cloud server and client have been presented and compared with other solutions. In addition, a number of connectivity options for file management were enumerated.

Inhaltsverzeichnis

1. Einleitung

Die deutsche Schul-Cloud soll “dabei helfen, die digitale Transformation in Schulen zu meistern und den fächerübergreifenden Unterricht mit digitalen Inhalten zu bereichern” [1]. Ein Schwerpunkt ist hierbei das Arbeiten mit verschiedenen Dateiformaten im Unterrichtskontext. Dazu soll die Schul-Cloud eine Möglichkeit schaffen, eigene Dateien zu verwalten und sie unter Lehrern und Schülern zu teilen. Die Bildung einer Dateiablage ist nach neueren Analysen fester Bestandteil von “technisch-organisatorische Kernanforderungen"[2] für den digitalen Unterricht. Dafür soll keine neue Dateiverwaltungstechnologie entworfen werden, sondern auf bestehende Systeme zurückgeführt werden. Vielmehr soll es für den Nutzer der Schul-Cloud möglich sein, ein bestehendes Dateisystem weiter zu nutzen und in die Schul-Cloud zu integrieren. Außerdem ist es erforderlich, Dateien auf verschiedenen Systemen zu lagern, um eine flexible Architektur zu schaffen. Dies ist vor allem dann nötig, wenn große Datenmengen von sehr vielen Schulen, Schülern und Lehrern entstehen. Dann ist es wichtig, eine Lösung in Form eines verteilten File Storage zu finden.

In dieser Bachelor-Arbeit wird eine solche Architektur für ein verteiltes Dateiverwaltungssystem beschrieben und entworfen. Begonnen wird mit einer **Vorbetrachtung** (2), in welcher sich über die bestehende Situation von Dateiverwaltung im Schulunterricht auseinander gesetzt wird. Auch werden bereits bestehende Konzepte anderer Arbeiten analysiert und Vorbilder ausgemacht. Danach wird ein **Konzept** (3) für ein verteiltes System beschrieben und anschließend eine mögliche **Implementierung** (4) aufgezeigt. In dieser wird eine Anbindung von File Storage-Providern im Schul-Cloud Server und Client beschrieben. Diese Anbindungsmöglichkeit wird anschließend **evaluiert** und mögliche **Anschlussszenarien** aufgezeigt (5). Zum Ende werden die Ergebnisse **zusammengefasst** (6).

[1] Technischer Bericht (Seite 5) [4]

2. Vorbetrachtung und Verwandte Arbeiten

Als erstes gilt es, gegenwärtige Technologien zu evaluieren, um eine solide Grundlage für eine verteilte Dateiverwaltung für die Schul-Cloud zu erstellen. Wie in der Einleitung erwähnt wurde, soll kein neuartiges Dateiverwaltungskonzept geschaffen werden. Vielmehr soll auf bestehenden Systemen und Technologien aufgebaut werden und deren Vorteile genutzt werden. Zu Beginn werden im Vorfeld formulierte Anforderungen aufgegriffen und analysiert. Hierzu diente der Technischen Bericht der Schul-Cloud [4] als Grundlage, welcher in Zusammenarbeit des Hasso-Plattner-Instut mit dem Mint-EC enstand. Ebenfalls werden andere Konzepte zur Dateiverwaltung im Schul-Kontext durchsichtet und analysiert. Es schließt sich eine Beobachtung an, inwieweit Dateiablagen auf Basis von Cloud-Storage Providern bereits im Unterricht genutzt werden. Für eine abschließende Vorbetrachtung eignet sich eine Betrachtung bereits bestehender Cloud-Storage Anbieter, welche sich in der Praxis bewährt haben und weiträumigen Einsatz finden.

2.1. Anforderungen an die Dateiverwaltung der Schul-Cloud

2.1.1. Die Cloud für Schulen in Deutschland: Konzept und Pilotierung der Schul-Cloud

Der Technische Bericht der Schul-Cloud [4] von (Meinel et. al) wurde von Wissenschaftlern am Hasso-Plattner-Institut in Potsdam geschrieben und beschäftigt sich mit einer ersten technischen Konzeption der Schul-Cloud. Dieser bildet die Grundlage für das Bachelor-Projekt der Schul-Cloud im Wintersemester 2016/17 und Sommersemester 2017, wessen sich diese Arbeit anschließt. Der Bericht beinhaltet im Kapitel 4.5 auch ein Konzept für das Dateimanagement. Dort wird beschrieben, dass Dateien auf allen Endgeräten verfügbar sein sollen. Außerdem wird ein Aufbau skizziert, auf den im weiteren Verlauf dieser Arbeit eingegangen wird (Kapitel 3). Weitere wichtige Aufgaben der Dateiverwaltung sollen das Speichern von “zusätzlichen Metadaten, die Erzeugung von Thumbnails und die Verwaltung von Berechtigungen"[4] sowie das Teilen von Inhalten

sein.

2.1.2. Studie „Bildung 2.0 - Digitale Medien in Schulen“

Sinnbildlich für viele Studien im Bereich digitaler Bildung, zeigt dieser Bericht der BITKOM [3] auf, dass die Bereitschaft von Schülern und Schülerinnen, von physischen zu digitalen Lerninhalten zu wechseln, stetig wachse. Immer öfter diene ein Computer als Hilfe für die Bearbeitung von Hausaufgaben. Dies Zeigt auch die Abbildung 1. Auch kämen PCs öfter im Unterricht zum Einsatz. Allerdings würde die Qualität der IT-Infrastruktur nicht immer als positiv wahrgenommen. Der Wunsch nach besserer Ausstattung und intensiverem Nutzen von elektronischen Medien sei klar ersichtlich. Somit scheint eine Verbindung der Dateiverwaltung der Schul-Cloud mit dem Hausaufgaben - Dienst und Kursen sowie Fächern äußert sinnvoll, weshalb sich in dieser Arbeit vermehrt darauf bezogen wird.

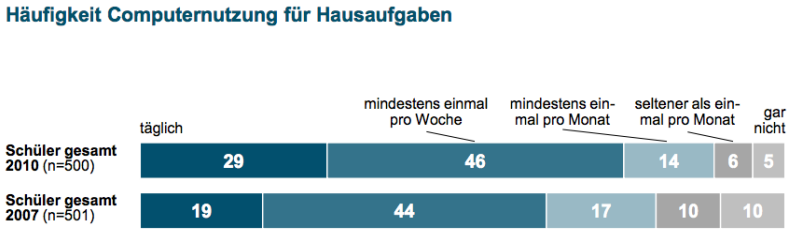

Abbildung 1: Häufigkeit der Computernutzung für Hausaufgaben im Vergleich von 2007 zu 2010[2]

2.1.3. Conceptual Architecture Design and Configuration of a Peer to Peer File System for Schools and Medium Sized Business

Damian Clarke evaluierte in seiner Arbeit den Einsatz eines verteilten Dateisystem für Schulen und kleine Firmen [5]. Die Idee ist es, weg von einem üblichen zentralen Dateiserver zu einem verteilten Peer-to-Peer [3] zu wechseln, um

[2]Studie „Bildung 2.0 - Digitale Medien in Schulen“ (Folie 14) [3]
[3]Peer-to-Peer - Verbindung - `https://de.wikipedia.org/wiki/Peer-to-Peer`

so eine bessere Auslastung und ein skalierbares System zu erzielen. Gerade in der Schullandschaft sei es von Vorteil, auf eine verteilte Struktur zu setzen, da die Masse an Daten stetig zu nähme und die Kosten für Wartung und Erweiterung auf mehrere Träger besser zu verteilen wären. Dies ist besonders wichtig im Hinblick auf die Masse an Schulen in der Schul-Cloud. Wo zu Beginn noch 26 Partner-Schulen auf dem Produktivsystem unterwegs sind, werden dies mit Ausweitung nach der Pilotphase immer mehr. Somit macht eine Verteilung der Dateibestände auf mehrere Server Sinn.

2.2. Nutzung von Cloud-Storage Providern im Unterricht

Die Verwaltung von eigenen und geteilten Dateien wird mit der zunehmenden Digitalisierung des Unterrichts immer wichtiger. Wo es früher reichte, seine Unterrichtsmaterialien in einem Schulhefter zu legen, will man Arbeitsblätter und Wissenstexte heute überall und immer verfügbar haben. Um einen Eindruck in diese Problematik zu bekommen, wurde eine Umfrage zum Thema 'Dateiorganisation in der Schule' [8] erstellt und an ehemaligen bzw. aktuellen Schülern sowie Lehrern geschickt. Insgesamt nahmen ***65*** Teilnehmer an dieser Umfrage teil. Gefragt wurde unter anderem nach der Rolle des Teilnehmers (Schüler oder Lehrer, Frage 1). Dabei sollten ehemalige Schüler und Lehrer vergangene Erfahrungen in die Beantwortung der Frage einfließen lassen. Dann wurde danach gefragt, ob der Befragte webbasierte Anwendungen zur Verwaltung der Dateien im Schulalltag benutzt (Frage 2). Anschließend wurde die Befragung geteilt. Wenn die Frage 2 mit 'Nein' beantwortet wurde, sollten Gründe für das Nichtbenutzen aufgezählt werden (Frage 3). Wurde Frage 2 mit 'Ja' beantwortet, wurde nach Vorteilen der Nutzung gefragt (Frage 4), sowie nach Beispielen, welche Dienste genutzt wurden (Frage 5).

Anpassen

Die Ergebnisse [7] dieser Umfrage zeigen, dass eine Mehrheit der Befragten (ca. 72%) auf solche Anbieter in ihrem Schulalltag verzichtet haben. Ein Großteil dieser Mehrheit gab an, dass Datenschutz ein großes Problem dabei darstellte. Ausländische Anbieter wie Dropbox [4] oder Google Drive [5] werden im pädago-

[4] Dropbox - `https://www.dropbox.com/de/?landing=cntl`
[5] Google Drive - `https://www.google.com/intl/de_ALL/drive/`

gischen Bereich eher kritisch betrachtet, da sie nicht in Deutschland gehostet werden und somit keine Transparenz in der Nutzung anfallender Daten erfolgt. Auch sind solche Dienste in Lernplattformen eher verpönt, da “Schulen keinen Einfluss auf die Verwendung und Weiternutzung der hier anfallenden Daten und haben keine Vertragsbeziehung mit dem Betreiber dieser Dienste"[6] haben. Weitere Gründe für die Nichtnutzung sind unter anderem, dass weite Teile des Unterrichtsmaterials offline zur Verfügung stehen, wie zum Beispiel einfache Arbeitsblätter, welche gar nicht erst digitalisiert werden. Oft ist auch einfach die Bereitschaft auf Seiten von Lehrer und Schüler gar nicht vorhanden, ihre Dateien online verfügbar zu machen, da die Ausstattung an den Schulen ohnehin auf einen weniger digitalen Unterricht ausgerichtet ist. Dazu kommen veraltete Technologien und schnelles Internet. Somit besteht auch kein Interesse daran, diese Dateien nur für den Gebrauch zu Hause verfügbar zu machen. Wenn dann doch Dateien von zu Hause in die Schule gebracht werden müssen, zum Beispiel Präsentationen oder Videos, wurden diese einfach auf einem USB-Stick gelagert.

Ein kleiner Teil der Befragten (ca. 28 %) nutzten oder nutzen dagegen bereits Cloud-Storage Dienste für ihre Dateiverwaltung. Als meistgenutzter Dienst wurden hier Google Drive, One Drive [6] und Dropbox genannt. Ein sehr kleiner Teil benutzte sogar selbstgehostete Dienste wie zum Beispiel eine ownCloud [7]. Die Gründe hierfür seien die stetige Verfügbarkeit von mehreren Geräten, die papierlose Nutzung von Unterrichtsmaterialien, ein leichterer Backup von wichtigen Dateien und das Teilen mit mehren Personen.

Die Auswertung der Umfrage zeigt, dass bei einem Großteil die Nutzung von cloudbasierten File-Storage Diensten eher auf Kritik stößt, als sie wirklich genutzt werden. Vor allem sind Datenschutz und schlechter Umgang mit digitalen Medien im Unterricht die Gründe hierfür. Mit einer einfachen Lösung innerhalb der Schul-Cloud, schnell an nötige Dateien für den Unterricht oder Hausaufgaben in digitaler Form zu kommen, könnte man hier ein Umdenken umleiten und die Vorteile sichtbar machen.

[6] One Drive - `https://onedrive.live.com/`
[7] ownCloud - `https://owncloud.org/`

2.3. Bestehende Cloud-Storage Provider

Die Dateiverwaltung der Schul-Cloud soll nicht konzeptionell von Grund auf neu erstellt werden. Vielmehr soll auf bereits gut funktionierende und weit verbreitete Dienste zurückgegriffen werden. Auch soll es möglich sein, dass eine Schule selbst die Art des Hostings bestimmen kann. Somit muss die Dateiverwaltung mehrere Typen einbinden können. Dies wird im Kapitel 3 als Form des *Strategy-Pattern*s eingeführt, wodurch es möglich ist, Adapter für bestehende Anbieter zu schreiben. Im Folgenden werden solche Anbieter betrachtet, welche als erstes in die Schul-Cloud integriert wurden und werden sollen.

2.3.1. AWS S3

Als Teil der im Jahre 2006 eingeführten Amazon Web Services [8] (AWS) bietet Amazon S3 (Simple Storage Service) als skalierbares Speichersystem zur Verfügung. S3 tritt hierbei als cloudbasierter Objektspeicher auf, der mit *Buckets* und *Objects* als Grundeinheiten arbeitet. Ein Bucket dient als einer Art Container, in dem mehrere Objects abgelegt werden. Ein Object ist somit eine Datei, die über einen *Key* eindeutig in einem Bucket referenziert ist. In diesen Keys spiegelt sich auch die Ordnerstruktur innerhalb eines Bucket bzw. einen S3-Servers wieder. Dieser Aufbau hat sich als Quasi-Standard bei Storage Providern etabliert, wodurch sich die Dateiverwaltung der Schul-Cloud diesen Standard zu Nutze machen kann.

2.3.2. ownCloud

Die ownCloud ist eine Open Source Technologie zum Verwalten von Dateien. Sie wird oft dazu benutzt, um Dateien über mehrere Geräte synchronisieren. Für die Synchronisierung der Dateiablage greift ownCloud auf WebDAV zurück [9]. Somit bringt eine ownCloud den Vorteil mit sich, dass man darauf existierende Dateiablagen wie eine lokale Dateiverwaltung benutzen kann. Aufgrund des

[8] Amazon Web Services - `https://aws.amazon.com/de/`

[9] Web-based Distributed Authoring and Versioning (WebDAV) - `https://de.wikipedia.org/wiki/WebDAV`

Open Source Status wird es oft als selbstgehostetes System an Schulen und Einrichtungen benutzt. Man erwartet sich hier maximalen Datenschutz, da keine Daten an kommerzielle Anbieter weitergereicht werden müssen. Dadurch ist die ownCloud ein interessantes Objekt für die Anbindung in der Schul-Cloud, da man selbst als Open Source Projekt auftritt.

2.3.3. Dropbox

Dropbox tritt seit 2007 als kommerzieller Filehosting-Dienst auf. Neben der Funktion als Cloud Storage Provider bietet es außerdem das Teilen von Dateien mit mehreren Nutzern. Ähnlich wie ownCloud hat man hier die Möglichkeit, über einen Synchronisations-Client die in der Cloud abgespeicherten Dateien auf lokalen Rechnern zu bedienen. Die Umfrage aus Abschnitt 2.2, dass die Benutzung von kommerziellen Anbietern wie Dropbox kritisiert wird. Somit wird es keine direkte Dropbox-Anbindung in der Schul-Cloud geben. In Sachen Nutzerfreundlichkeit und Bedienbarkeit hat sich die Dateiverwaltung der Schul-Cloud jedoch ein Vorbild genommen.

3. Konzept

In den vorangegangenen Kapiteln wurde aufgezeigt, was die Dateiverwaltung der Schul-Cloud leisten muss und woran sie sich orientieren kann. Vom Partner des Bachelor-Projekts MINT-EC [10] und dem Hasso-Plattner-Institut wurde eine Anforderungsspezifikation erstellt, an welcher sich das folgende Konzept der Dateiverwaltung orientiert (Abbildung 2). Neben der Verwaltung von eigenen Dateien, sollen auch jene im Kurs- bzw. Fächer- so wie Klassenkontext verteilbar sein. Dies soll dazu dienen, digitale Inhalte besser in den Unterricht einzubetten und (Haus)-Aufgaben mithilfe von anschaulichem Material zu gestalten. Somit sollen diese Dateien über eine einheitliche Schnittstelle durch die Backend-API sowie über das Web-Frontend in mehreren Kontexten verfügbar sein. Außerdem soll es für eine Schule möglich sein, eine bereits bestehende Dateiablage einzubinden.

Abbildung 2: Grundanforderungen an die Dateiverwaltung der Schul-Cloud[11]

3.1. Grundaufbau Dateiverwaltung

Die grundlegende Struktur sieht vor, dass jede Schule seine eigene logische Einheit besitzt. Diese wird im folgenden *Bucket* genant und leitet sich vom AWS S3-Standard ab (Abschnitt 2.3.1). Das dient zum einen der besseren Organisation

[10] Verein mathematisch-naturwissenschaftlicher Excellence-Center an Schulen e. V. (MINT-EC) - `https://www.mint-ec.de/`

[11] Martin Hense (Mint-EC - `https://www.mint-ec.de/`)

innerhalb der Schul-Cloud, da Nutzer im Datenmodell zu einer Schule zusammengefasst werden. Außerdem folgt das Projekt dem Prinzip *Privacy by Design* [12]. Damit wird sichergestellt, dass eine Schule außerhalb von anderen Schulen modular abgekapselt wird. Somit ist von vornherein eine Sicherheit der Dateien auf Schulebene gewährleistet. Diese Schul-Buckets können auf getrennten Servern liegen, so dass zum Beispiel eine Schule aus Niedersachsen auf eine eigene ownCloud zurückgreifen und eine Schule aus Hamburg einen S3-Provider benutzen kann. Das Prinzip des *File Sharing* (Abschnitt 3.4) kann diese logische Abkapselung ein wenig aufbrechen, um Dateien auch über Schulebene hinaus teilbar zu machen. Eine Erweiterung von Buckets auf Landkreis - oder Bundeslandebene erscheint als weitere Option. Jedoch wird sich diese Arbeit auf die Einteilung in Schul-Buckets beschränken. Dieses Konzept ist aber ohne weiteres skalierbar auf größere Kontexte. Die Abbildung 3 zeigt im folgenden die weitere Unterteilung eines Buckets. Dieser wird in vier Teilkomponenten innerhalb der Schule untergliedert. Wenn ein Bucket als einer Art Oberordner oder Root-Verzeichnis verstanden werden kann, dann sind diese Teilkomponenten schlichtweg Unterordner des Buckets. Zum einen gibt es den Ordner *courses*, dieser enthält alle Dateien, welche zu Kursen bzw. Fächer gehören. Für eine bessere Unterteilung befinden sich hier sehr viele Unterordner, welche bloß die *courseId* eines Schul-Cloud Kurses referenzieren. Dies dient der Verteilung aller Kurs-Dateien zu dem zugehörigen Kurs. Diese ID-Ordnernamen werden von Schul-Cloud Server versteckt und nicht im User Interface des Clients angezeigt. So hat man aber zum Beispiel die Möglichkeit, über die *courseId* alle Dateien eines Kurses zu erhalten. Auf diese haben nur Lehrer und Schüler des Kurses bzw. Faches Zugriff. Die Berechtigungsverwaltung übernimmt der Schul-Cloud Server und wird in der Implementierung (Abschnitt 4.1.3) genauer beschrieben.

Ähnlich wie der *courses* Unterordner gibt es den *classes* und *users* Unterordner. Beim ersten werden im gleichen Schema alle Dateien von Schul-Cloud Klassen verteilt, wieder referenziert über den jeweiligen ID-Unterordner. Zugriff haben hier nur Lehrer und Schüler der Klasse. Der *users* Unterordner bildet die per-

[12] Privacy by Design - `https://digitalcourage.de/blog/2015/was-ist-privacy-design`

sönlichen Dateien eines jeden Nutzers der Schule, egal ob Lehrer, Schüler oder Administrator. Als Zusatz für schulübergreifende Dateien gibt es den Unterordner *school*. Schreibrechte hat hier nur der Administrator der Schule, zugreifen kann jedoch jeder Nutzer der Schule. Natürlich kann es nun vorkommen, dass eine Schule bereits eine ausgeprägte Dateistruktur besitzt und der Grundaufbau mit den vier Unterkomponenten nicht abbildbar sein könnte. Hier muss der Schul-Cloud Server dafür sorgen, dass diese Dateien trotzdem über die Schul-Cloud zugänglich sind. Eine Möglichkeit wäre es, dass die gesamte Struktur zu Beginn in den *schools* Ordner gelegt wird und diese von dahin auf die jeweiligen Kontexte verlegt werden.

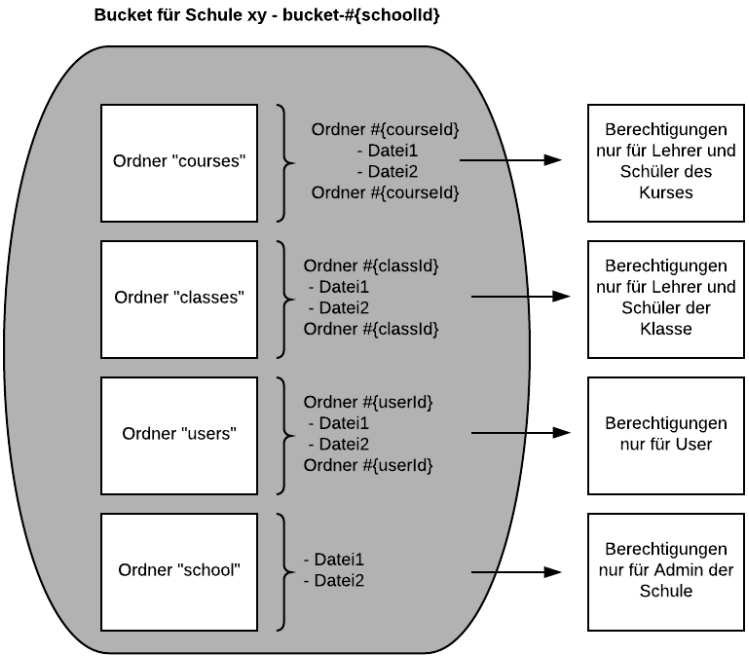

Abbildung 3: Grundaufbau eines Schul-Buckets

3.2. Architektur verteilter Provider

Die Schul-Cloud stellt während der Pilotphase zwar nur eine S3-Instanz [13], worauf die Buckets der Pilotschulen liegen. Trotzdem soll es die Architektur der Dateiverwaltung möglich machen, Buckets auch auf mehrere Server zu verteilen. Dafür macht sich der Server das Strategy-Pattern [14] zu Nutze. Dieses sieht vor, dass eine abstrakte Strategie die Schnittstellen vorgibt und mehrere Strategien diese implementieren. In Form dieser können so Anbindungen an verschiedene Typen von File Storage Providern erstellt werden. In der Implementierung wird eine solche Strategie am Beispiel von AWS S3 (Abschnitt 4.1.2) geschildert. Abbildung 4 zeigt die Verteilung der verschiedenen File Storage Strategien. So ist es möglich, über den Schul-Kontext, der die Art des Provider bestimmt, die richtige Strategie zu bestimmen und auf die Schuldateien zugreifen zu können. Diese Aufteilung macht die Verteilung von Buckets auf mehrere Server ausführbar. So kann ein Bucket auf einer S3-Instanz liegen und mittels S3-Strategie darauf zugegriffen werden, sowie eine ownCloud-Instanz durch die ownCloud-Strategie auf einem anderen Server. Zwar ähnelt die genannte Struktur sehr dem S3-Ansatz, er lässt sich jedoch auch auf andere Typen einsetzen. Ein Bucket im ownCloud-Kontext kann hierbei ein normaler ownCloud-Ordner sein, der durch bestimmte Sicherheitsvorkehrungen nur für den Schul-Administrator zugänglich ist.

[13] Technischer Bericht der Schul-Cloud, Seite 39 [4]
[14] Strategy-Pattern - `https://de.wikipedia.org/wiki/Strategie_(Entwurfsmuster)`

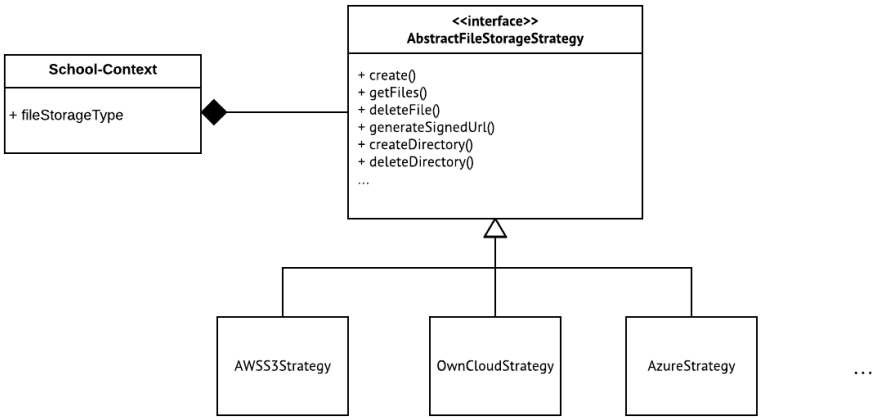

Abbildung 4: Verteilung der Schul-Buckets mithilfe des Strategy-Patterns

3.3. Interaktion verschiedener Schul-Cloud Komponenten

Bilder Qualität verbessern!!

Im Folgenden wird die Interaktion der verschiedenen Schul-Cloud Komponenten mithilfe von Beispielen beschrieben. Es wird stets von drei Rollen der Kommunikation geredet. Zum einen sorgt der *Client* für die Nutzeraktion und steht somit für das Schul-Cloud Frontend, aber auch für eine mobile App. Der *Server* meint den Schul-Cloud Server und dient als einer Art Proxy zu den verschiedenen File-Storage Providern. Dieser wird im Folgenden als *Provider* bezeichnet. Als erstes Beispiel soll das Holen der persönlichen Dateien dienen. Abbildung 5 zeigt die Interaktion der drei Komponenten. Der Client beginnt hierbei mit einem GET-HTTP-Request an die */fileStorage/* - Route vom Server. Um Dateien für einen bestimmten Kontext zu bekommen, muss im Query-Parameter *path* der richtige Pfad mitgegeben werden. Für die persönlichen Dateien des Nutzers mit der ID *123* ist dies *users/123*. Man sieht hierbei die Rückführung auf den Grundaufbau der Dateiverwaltung. Die persönlichen Dateien eines jeden Schul-Cloud Nutzers werden im *users* Unterordner der zugehörigen Schule gespeichert. Der ID-Ordner referenziert hier die Dateien des gesuchten Nutzers mit der ID *123*. Dies muss eine valide Nutzer-ID sein, welche in der Schul-Cloud Datenbank gespeichert wird. Der Server kümmert sich nun um die Berechtigung, in dem er

zum einen prüft, ob der im Client angemeldete Nutzer auch auf den Pfad zugreifen darf. Im Detail wird dies noch im Abschnitt 4.1.3. beschrieben. Der Server ermittelt die Schule des Nutzers und fragt dann beim richtigen Provider die gesuchten Dateien für den gegebenen Pfad an. Dieser gibt dann die Datei-Objekte aus, welche dann über den Server zum Client ausgegeben werden. Man sieht hier sehr gut, dass der Schul-Cloud Server eine verwaltende Rolle einnimmt, ohne die Dateien tatsächlich physisch zu speichern. Er sorgt vielmehr für die Verteilung der Requests an die richtigen File Storage Provider. Dafür nutzt er das zuvor dargestellte Strategy-Pattern, um für eine Schule den richtigen Bucket zu finden.

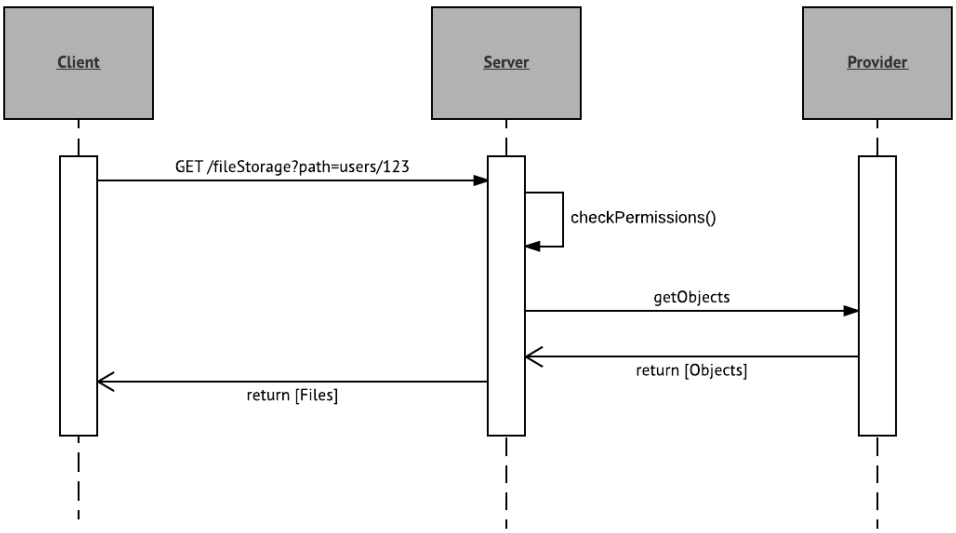

Abbildung 5: Interaktion zwischen Client, Server und File Storage Provider beim Holen von persönlichen Dateien

Neben dieser einfachen dreigeteilten Interaktion zwischen Client, Server und Provider werden Upload und Download von Dateien direkt zwischen Client und Provider gehandhabt. Das hat den Grund, dass die Übertragung großer Datenmengen bei Dateien im Gigabyte - Bereich nicht über den Server laufen sollen. Abbildung 6 zeigt dies anhand des Uploads einer Datei. Der Server fragt beim Provider nun eine Upload-URL an, auf welche der Client dann die Datei

example.jpg hochladen kann. Der Query-Parameter *action* teilt mit, um welchen Typ von URL es sich handelt, im Falle von *putObject* beim Upload einer Datei. Der Server prüft wieder die Berechtigungen des Clients und sendet dann die URL an den Client zusammen mit wichtigen Meta-Daten der Datei zurück. Diese soll der Client zusammen mit der Datei zum Provider laden, damit der Server später mehr Informationen über die Datei hat, zum Beispiel die Referenzierung eines Vorschaubilds.

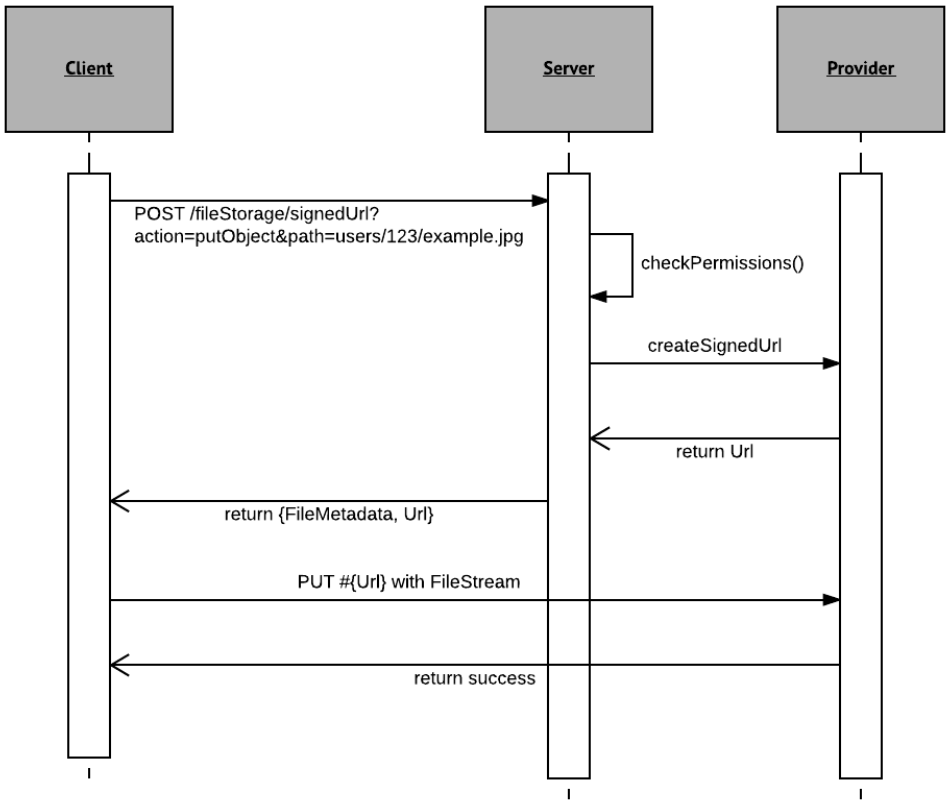

Abbildung 6: Interaktion zwischen Client, Server und File Storage Provider beim Upload einer Datei

3.4. Teilen von Dateien

Zwar gibt der Aufbau der Dateiverwaltung eine klare Aufteilung zwischen Schul-, Kurs-, Fach - und Klassenkontexten vor, es soll jedoch trotzdem möglich sein, Dateien mit anderen Nutzern zu teilen. Das ist vor allem dann von Vorteil, wenn ein Lehrer eine persönliche Datei in seinem Unterricht direkt benutzen möchte. Zwar könnte der Lehrer dann die Datei in den jeweiligen Kursordner hochladen bzw. direkt im Unterrichtsmodus des Clients hochladen; welcher im Abschnitt 4.2 beschrieben wird; manchmal möchte der Lehrer trotzdem seine eigenen Dateien bei sich behalten und die Freigabe selbst steuern können. Ein weiteres Beispiel wäre es, wenn ein Schüler eine private Datei mit anderem teilen möchte, ohne sie im Kurskontext verfügbar zu machen. Somit muss es möglich sein, im Schul-Cloud Client einen Zugriffslink zu generieren. Abbildung 7 zeigt den Prozess zur Generierung eines solchen Links für eine Datei. Der Client fordert die Freischaltung für das Teilen der Datei beim Server an. Dieser besteht intern aus dem eigentlichen File-Service und aus dem Link-Service, welcher als URL Shortener [15] auftritt. Dieser dient dazu, den langen Link zu einer Schul-Cloud Datei etwas zu verkürzen und leichter teilbar zu machen. Nachdem der File-Service das Teilen freigegeben hat, fragt der Client einen solchen Shortlink an und gibt diesen im User Interface an. Dieser Prozess ist bekannt aus bereits bestehenden File Storage Providern.

[15] Kurz-URL-Dienst - `https://de.wikipedia.org/wiki/Kurz-URL-Dienst`

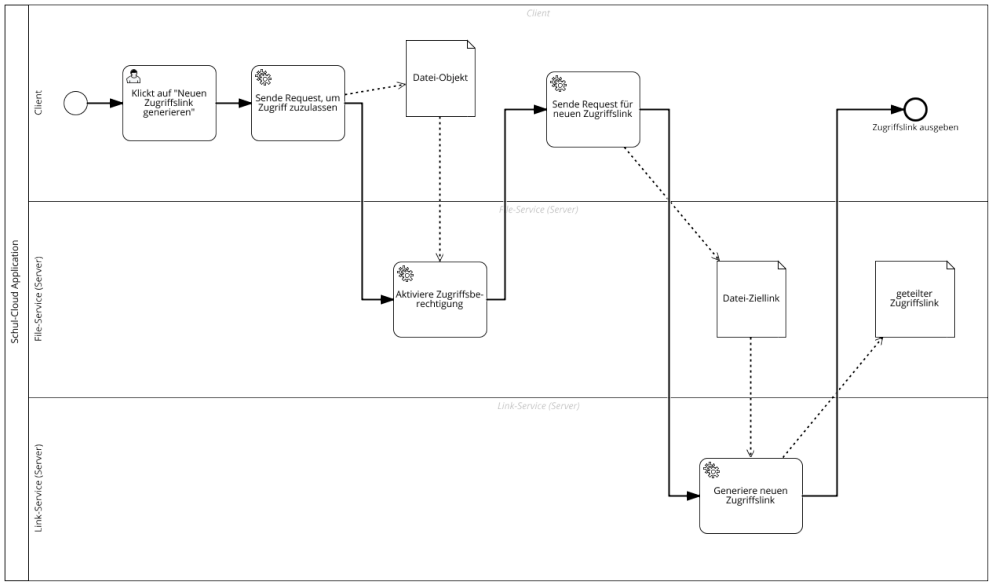

Abbildung 7: Generierung eines Zugriffslinks

Diesen Link kann der Besitzer der Datei nun an andere Schul-Cloud Nutzer weiterleiten. Den daraus resultierenden Zugriff ist in Abbildung 8 modelliert. Der Client öffnet den Link und fragt zuerst beim Server an, ob die Sharing-Funktion für die Datei überhaupt freigeschaltet ist. Ist dies nicht freigegeben, erhält der Client einen Fehler. Im anderen Falle wird im Server gespeichert, welcher Nutzer auf die Datei zugreifen möchte, um den Prozess später zu beschleunigen und redundante Prüfungen zu verhindern. Außerdem können so für den Nutzer freigegebene Dateien zusammen aufgelistet werden, wie es Dropbox beispielsweise tut. Anschließend wird eine Anfrage auf die tatsächliche Datei über den Server an den Storage Provider gestellt und diese zurück an den Client gesendet. Dieser Teil des Prozesses ist sehr vereinfacht dargestellt, tatsächlich muss für das Holen der tatsächlichen Datei zunächst eine Zugriffs-URL, wie in Abschnitt 3.3 beschrieben, generiert werden und an den Client geliefert werden.

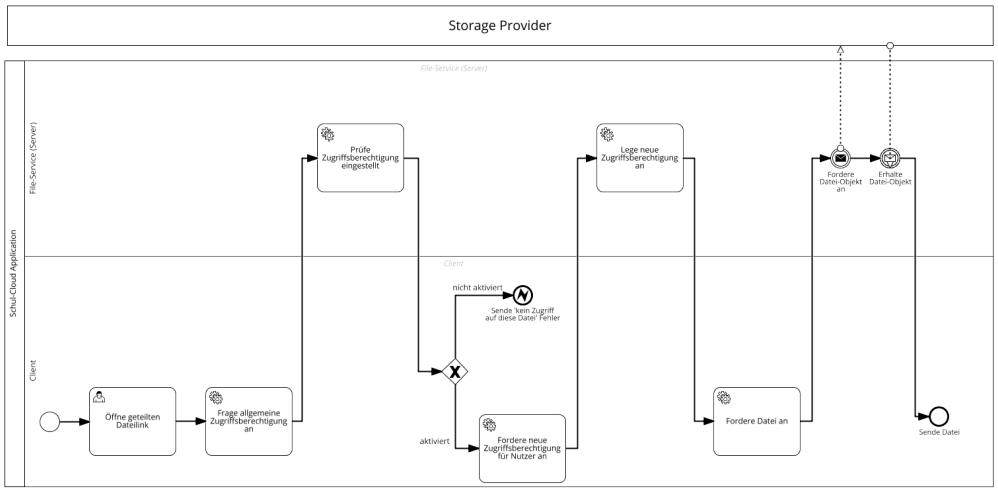

Abbildung 8: Zugriff auf eine geteilte Datei

4. Implementierung

Nachdem das grobe Konzept der Dateiverwaltung der Schul-Cloud geschildert wurde, werden anschließend Implementierungsdetails präsentiert. Diese wird in einer Zweiteilung erfolgen. Zunächst wird relevanter Code des Schul-Cloud Servers [16] aufgezeigt und beschrieben. Dann werden dazu passende Teile vom User Interfaces des Schul-Cloud Clients [17] dargelegt. Beide Komponenten sind öffentlich auf GitHub [18] zugänglich.

4.1. Schul-Cloud Server

4.1.1. Einbettung in das Schul-Cloud - Datenmodell

Der Schul-Cloud Server basiert auf Node.js [19] und Feathers [20], einer auf Express [21] aufbauenden Schicht für das schnelle Entwickeln von REST-APIs [22]. Die Dateiverwaltung des Backends wird somit auch als typischer Feathers-Service bereitgestellt. Dieser befindet sich im Unterordner *src/services/fileStorage* und beinhaltet die grundlegende Funktionalität der Dateiverwaltung. Anders als typische Feathers Services besitzt dieser kein MongoDB [23] - Model, sondern dient eher als Proxy-Service. Die *index.js* bündelt die drei Teilservices des File-Storage Services zusammen und macht sie nach außen zugänglich (Anhang 2). Diese sind zum einem der *FileStorageService*, welcher sich um die eigentliche Verteilung von Dateianfragen, wie das Erstellen und Löschen von Dateien, kümmert. Dazu kommt der *SignedUrlService*, welcher die Zugriffslink generiert. Als drittes gibt es den *DirectoryService*, welcher zusätzliche Funktionalitäten für Ordner bereit stellt. Alle API-Routen werden unter */fileStorage/* registriert, wobei der *Si-*

[16] Schul-Cloud Server - `https://github.com/schul-cloud/schulcloud-server`
[17] Schul-Cloud Client - `https://github.com/schul-cloud/schulcloud-client`
[18] GitHub - `https://github.com`
[19] Node.js - `https://nodejs.org`
[20] Feathers - `https://feathersjs.com/`
[21] Express - `http://expressjs.com/de/`
[22] Representational State Transfer (REST) - `https://de.wikipedia.org/wiki/Representational_State_Transfer`
[23] MongoDB - `https://www.mongodb.com/de`

gnedUrlService unter */fileStorage/signedUrl* und der *DirectoryService* unter */fileStorage/directories* verfügbar ist. Die einzelnen Funktionen aller API-Routen sind in der Dokumentation des Servers [1] verfügbar.

In der *index.js* des FileStorage Services erfolgt außerdem die Auswahl der richtigen Strategie für die Schule des Zugreifenden auf die API. Eine in der SchulCloud vermerkte Schule kann genau einen *fileStorageType* besitzen, wie die *model.js* des SchoolServices (*src/services/school*) des Backend zeigt:

```
1  'use strict';
2
3  const fileStorageTypes = ['awsS3']; // aktuell unterstützte Strategien
4
5  const schoolSchema = new Schema({
6    name: {type: String, required: true},
7    address: {type: Object},
8    fileStorageType: {type: String, enum: fileStorageTypes}, // Eigenschaft zur
         Auswahl der richtigen Strategie
9    systems: [{type: Schema.Types.ObjectId, ref: 'system'}],
10   federalState: {type: Schema.Types.ObjectId, ref: 'federalstate'},
11   createdAt: {type: Date, 'default': Date.now},
12   updatedAt: {type: Date, 'default': Date.now}
13 } ,{
14   timestamps: true
15 });
```

Vor dem Aufrufen jeder Route wird die Funktion *resolveStorageType* in *src/services/fileStorage/hooks/index.js* ausgeführt, die dafür sorgt, dass der Typ des SchulBuckets aus der im Nutzer referenzierten Schule erhalten wird. Die *userId* wird vorher wie bei Feathers üblich aus dem Request via Authentifizierung ermittelt.

```
1  const resolveStorageType = (hook) => {
2    let userService = hook.app.service("users");
3
4    // finde angemeldeten Nutzer
5    return userService.find({query: {
6        _id: hook.params.payload.userId,
```

```
7         $populate: ['schoolId'] // hole referenzierte Schule
8       }}).then(res => {
9
10        // speichere korrekten Typ
11        hook.params.payload.fileStorageType = res.data[0].
              schoolId.fileStorageType;
12        return hook;
13    });
14  };
```

Aus dieser Information kann der FileStorage Service dann die richtige Strategie wählen. Hier ein Beispiel für die Funktion *find*, welche auf *GET /fileStorage/* registriert ist und die Dateien für einen gegebenen Pfad holt:

```
1   /**
2   * @returns {Promise}
3   * @param query contains the file path
4   * @param payload contains fileStorageType and userId, set by middleware
5   */
6   find({query, payload}) {
7     return createCorrectStrategy(payload.fileStorageType).getFiles(payload.userId
          , query.path);
8   }
```

Die Funktion *createCorrectStrategy* erzeugt ein Objekt für die jeweilige Strategie, welche in *src/services/fileStorage/strategies* implementiert sind:

```
1   const strategies = {
2     awsS3: AWSStrategy
3   };
4
5   const createCorrectStrategy = (fileStorageType) => {
6     const strategy = strategies[fileStorageType];
7     if (!strategy) throw new errors.BadRequest("No file storage provided for this
          school");
8     return new strategy();
9   };
```

4.1.2. AWS S3 - Strategy als Beispiel

Im Schul-Cloud Server lassen sich eine Reihe von Strategien für verschiedene Typen von File Storage Providern implementieren. Diese werden im Ordner *src/-services/fileStorage/strategies* abgelegt. Die Klasse *AbstractFileStorageStrategy* (Anhang 1) gibt die zu implementierenden Funktionen vor. Die einzelnen Strategien sind somit Unterklassen dieser. In diesem Abschnitt wird die Implementierung für einen AWS S3-Adapter beschrieben. S3 dient während der Pilotphase der Schul-Cloud als Standart-Provider. Um schnell entwickeln zu können, wurde ein S3-Abbild auf dem Development-System der Schul-Cloud [24] eingestellt. Hierbei wurde *minion* [25] genutzt, einem Open Source Projekt, was Funktionen von S3 nachbildet. Die AWS S3-Strategy befindet sich in *src/services/fileStorage/strategies/awsS3.js*.

Um Zugriff auf eine S3-Instanz zu bekommen, brauch die Strategie eine AWS-S3-Konfiguration. Das Node-Modul *aws-sdk* [26] bietet hierbei eine komplette Anbindungsmöglichkeit. Eine mögliche S3-Konfiguration könnte folgendermaßen aussehen:

```
1
2   const awsConfig =  {
3     "signatureVersion": "v4",
4     "s3ForcePathStyle": true,
5     "sslEnabled": true,
6     "accessKeyId": "password",
7     "secretAccessKey": "password",
8     "region": "eu-west-1",
9     "endpointUrl": "https://localhost:3000"
10  }
```

Mithilfe dieser Konfiguration und dem aws-sdk kann man nun Befehle an eine S3-Instanz schicken. Der AWS S3-Standard unterstützt alle der in der *Ab-*

[24] Development-System - `https://schul.tech/`
[25] minion - `https://github.com/minio/minio`
[26] aws-sdk - `https://aws.amazon.com/de/sdk-for-node-js/`

stractFileStorageStrategy geforderten Funktionen. In Anhang 3 wird die Implementierung der Funktion *deleteFile* für das Löschen einer Datei in einem S3-Bucket gezeigt. In jeder Funktion der Strategie muss zuerst die Berechtigung geprüft werden. Dies übernimmt der *filePermissionHelper*, welcher im nächsten Abschnitt näher vorgestellt wird. Ein weiterer wichtiger Punkt beim Verbinden mit der S3-Instanz ist das Erstellen eines AWS S3-Verbindungsobjekts (Zeile 13). Dieser beinhaltet die vorher angelegte Konfiguration sowie die Information, mit welchem Bucket sich verbunden werden soll. Da die Architektur vorgibt, dass Buckets zu Schulen gehören, ergibt sich der Name des Buckets einfach aus *"bucket-"+ schoolId*. Die Funktion *createAWSObject* zeigt die Generierung dieses Objekts:

```
1   const createAWSObject = (schoolId) => {
2     if (!awsConfig.endpointUrl) throw new Error('AWS integration is not
          configured on the server');
3
4     // Verbinden der Konfiguration
5     var config = new aws.Config(awsConfig);
6
7     // Einstellen des S3-Endpunktes
8     config.endpoint = new aws.Endpoint(awsConfig.endpointUrl);
9
10    // Einstellen des gesuchten S3-Buckets
11    let bucketName = `bucket-${schoolId}`;
12
13    // Erstellen des S3-Objekts
14    var s3 = new aws.S3(config);
15
16    return { s3: s3, bucket: bucketName };
17  };
```

In der Variable *params* in Zeile 16 von Anhang 3 wird dann angegeben, um welche Art von Aktion es sich auf das Bucket handelt. Da in dieser Funktion die mit *path* angegebene Datei gelöscht werden soll, wird ein Array von zu löschenden Datei-Keys angelegt, welches die gesuchte Datei enthält. Eine S3-Datei ist immer über ihren Pfad referenziert. Es ist so theoretisch möglich, mehrere Dateien auf einmal zu löschen. Das macht sich die S3-Strategie beim Löschen von

Ordnern zu Nutze. Zum Schluss wird die Aktion auf dem Bucket ausgeführt. Dazu wird üblicherweise *awsObjekt.s3.deleteObjects(params)* genutzt. Um besser mit *Promises* [27] arbeiten zu können, wird *promisify* [28] um die Funktion gelegt. Der Vorteil des Strategy-Patterns ist es nun, dass die Anbindung an eine ownCloud-Instanz komplett anders implementiert werden kann. Man muss darauf achten, dass Ein- und Ausgabe-Parameter gleich sind. Hier macht sich der Status als Open Source Projekt bezahlt, da externe Entwickler verschiedene Strategien mit einbringen und so den File-Storage Service der Schul-Cloud beliebig erweitern können. Die S3-Strategie bietet hier einen sehr guten Einstiegspunkt für weitere Implementierungen.

4.1.3. Berechtigungsverwaltung

Vor jedem Aufruf einer Dateioperation werden die Berechtigungen des zugreifenden Nutzers geprüft. Wie in Anhang 3 (Zeile 5) zu sehen ist, kümmert sich der *filePermissionHelper* darum. Dieser befindet sich in *src/services/filePermission/utils/filePermissionHelper.js* und gehört zum *FilePermissionService*, einem weiteren Feathers Service, der sich ausschließlich um die Rechteverwaltung innerhalb der Dateiverwaltung kümmert. Die Funktion *checkPermissions()* ist folgendermaßen aufgebaut:

```
1  checkPermissions(userId, filePath, permissions = ["can-read", "can-write"]) {
2    return this.checkNormalPermissions(userId, filePath)
3      .catch(err => this.checkExtraPermissions(userId, filePath, permissions));
4  }
```

Wie hier zu sehen, teilt sich die Überprüfung in zwei einzelne Berechtigungschecks. Zum einen wird in der Funktion *checkNormalPermissions* die generelle Berechtigung überprüft, etwa ob es sich um eine eigene Datei handelt bzw. um eine Kursdatei und der zugreifende Nutzer Teilnehmer dieses Kurses ist. Diese Informationen erhält er aus dem *filePath*, da in diesem die Zugehörigkeit der

[27] Promises - https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Promise

[28] es6-promisify - https://www.npmjs.com/package/es6-promisify

Datei enthalten ist (siehe Abschnitt 3.1). Die komplette Implementierung dieses Checks findet sich in Anhang 4. Sollte diese Überprüfung fehlschlagen, gibt es noch die Möglichkeit, dass der Zugreifende die Datei von einem anderen Nutzer freigegeben bekommen hat. Dann muss ein Datenbankeintrag vom Typ *FilePermissionModel* für den Nutzer und die Datei existieren. Der *FilePermissionService* verwaltet dieses Datenbankmodell in der Datei *src/services/filePermission/model.js*:

```
1  const permissionTypes = ['can-read', 'can-write'];
2
3  const filePermissionSchema = new Schema({
4    key: {type: String, required: true, unique : true},
5    permissions: [{
6      userId: {type: Schema.Types.ObjectId, ref: 'user'},
7      permissions: [{type: String, enum: permissionTypes}]
8    }],
9    createdAt: {type: Date, 'default': Date.now},
10   updatedAt: {type: Date, 'default': Date.now}
11 });
```

Um eine zusätzliche Zugriffsberechtigung auf die Datei zu bekommen, muss ein solcher Eintrag in der Datenbank existieren. Die Eigenschaft *key* beinhaltet den Pfad der Datei. Im Array *permissions* sammeln sich Tupel aus Nutzer-ID und bestimmten Rechten. Das können Schreib- sowie Leserechte sein. Um die Menge an Datenbankeinträgen zu sparen, gibt es für jede Schul-Cloud Datei nur einen Eintrag, welche alle zusätzlichen Berechtigungen in *permissions* speichert. Somit kann man zum Beispiel überprüfen, ob eine Datei für einen bestimmten Nutzer zum Teilen freigegeben wurde. Falls es für eine Datei einen Datenbankeintrag gibt mit einem leeren *permissions* Array, gibt dies schon Auskunft darüber, ob die Datei überhaupt zum Teilen freigegeben ist, zum Beispiel wenn ein geteilter Link im Client erstellt wurde. Momentan ist nur die Freigabe für einzelne Nutzer möglich, so dass man bei Gruppenfreigaben jeden einzelnen Benutzer hinzufügen müsste. Dies lässt sich aber noch auf Gruppenebene erweitern. Mit diesen Datenbankeinträgen überprüft die Funktion *checkExtraPermissions()* (Anhang 5), ob eine zusätzliche Zugriffsberechtigung für den Nutzer und der ge-

gebenen Datei existiert. Führen beide Checks zu keinem positiven Ergebnis, hat der Nutzer definitiv keine Berechtigung für diese Datei.

4.2. Schul-Cloud Client

4.2.1. Anbindung zum Server

Der Schul-Cloud Client ist ebenfalls mit Node.js implementiert. Er kommt ohne großes Web-Framework aus und basiert auf einem einfachen Express-Server. Hier benutzt er *Handlebars* [29] als Template Engine. Die Anbindung an den Schul-Cloud Server erfolgt über den *filesController* in der Datei *controllers/files.js*. Hier gibt es eine Reihe von Dateioperationen, welche regelmäßig mit dem Schul-Cloud Server interagieren. Ein Beispiel ist das Erstellen eines neuen Ordners:

```
1  // erstelle neuen Ordner
2  router.post('/directory', function (req, res, next) {
3    const {name, dir} = req.body;
4
5    const basePath = dir;
6    const dirName = name || 'Neuer Ordner';
7
8    // Server-Anbindung
9    api(req).post('/fileStorage/directories', {
10     json: {
11       path: basePath + dirName,
12     }
13   }).then(_ => {
14
15     // Rückmeldung an den Browser
16     res.sendStatus(200);
17   }).catch(err => {
18     res.status((err.statusCode || 500)).send(err);
19   });
20 });
```

Für jeden Zugriff auf den Server dient das *api* - Objekt, welches in der Datei *api.js*

[29] Handlebars - `http://handlebarsjs.com/`

im Stammverzeichnis initialisiert wird. Dieser baut eine einfache Verbindung zum Schul-Cloud Server und macht HTTP-Requests möglich. Zum Erstellen eines Ordners muss man demzufolge *POST /fileStorage/directories/* aufrufen. Der *filesController* sorgt zudem für das Rendering der richtigen Template-Dateien.

4.2.2. User Interface

Im Folgenden wird auf die genaue Implementierung der einzelnen Dateioperationen im Client nicht näher eingegangen. Viel mehr liegt der Schwerpunkt auf die Nutzerinteraktion. Dazu dienen einzelne Screenshots vom Schul-Cloud Produktivsystem [30] (Stand 16. Juni 2017). Die Projektstruktur gibt es vor, dass sich das User Interface der Dateiverwaltung im Unterordner *views/files* befindet. Die normale Dateiverwaltung befindet sich auf der Route */files/* bzw. unter dem Menüpunkt "Meine Dateien". Hier sieht man zunächst die persönliches Dateien (Anhang Abbildung 9), d.h. alle diejenigen, die unter *üsers/"+ Nutzer-ID* des angemeldeten Nutzers gespeichert sind. In der Seitennavigation sieht man nun weitere Menüpunkte unter "Meine Dateien". So kann man nun auf die Kurs- sowie Klassendateien zugreifen. In dieser Dateiverwaltung hat man außerdem die Möglichkeit, die Datei im Browser zu öffnen. Ähnliche Funktionalitäten enthalten außerdem die Android [31] - und iOS [32] - App der Schul-Cloud.

Neben der normalen Dateiansicht für persönlichen, Kurs - und Klassendateien hat man außerdem die Möglichkeit, als Lehrer Dateien direkt im Unterrichtskontext einzufügen. Dazu geht man auf einen Kurs, dann auf ein Thema, bearbeitet dieses Thema und fügt eine Text-Komponente hinzu. Hier wurde der eingebaute *ckEditor* [33] so überarbeitet, dass man entweder per Drag & Drop eine Datei hineinschieben kann oder über die Bilderauswahl ein Bild aus dem Kurskontext auswählen kann (Anhang Abbildung 10).

[30] Schul-Cloud Produktivsystem - `https://schul-cloud.org`
[31] Schul-Cloud Android-App - `https://github.com/schul-cloud/schulcloud-mobile-android`
[32] Schul-Cloud iOS-App - `https://github.com/schul-cloud/schulcloud-mobile-ios`
[33] ckEditor - `http://ckeditor.com/`

4.2.3. Administration

Da es einer Schule möglich sein soll, selbst über die Art des File Storage Providers zu bestimmen, gibt es für die Rolle *Administrator* einen besonderen Menü-Punkt in der Seitenmenüleiste. Unter “Administration"kann der Schul-Administrator neben anderen Informationen auch den File Storage Typ auswählen (Anhang Abbildung 11). Hier soll es in Zukunft auch ein Import für bestehende Dateiablagen sowie die Einstellung für Berechtigungen sec:surve.

4.2.4. Dateien teilen

Das Teilen von Dateien funktioniert wie in Abschnitt 3.4 beschrieben momentan mit dem Generieren von Zugriffslinks. Dafür kann man auf das Teilen-Icon an einer Datei klicken und ein Modal-Fenster mit einem neuen Link öffnet sich (Anhang Abbildung 12). Dieser kann an mehrere Leute weitergegeben werden. Erst mit dem Klick auf das Teilen-Icon wird die Teilen-Funktion freigeschaltet.

5. Evaluierung und zukünftige Arbeit

Es gilt nun, die erarbeiteten Konzepte mit den Anforderungen abzugleichen. In Abschnitt 2 wurde klar, dass zum einen die Ablage von Dateien in mehreren Kontexten möglich sein soll. Dies wurde erreicht, indem man den Grundaufbau der Dateiverwaltung genau auf diesem Prinzip aufbauen lies. Die einzelnen Schul-Cloud Dateien sind genau einem Schul-Bucket zugeteilt und dort in den zugehörigen Kontexten, d.h. persönliche Dateien sowie Kurs - und Klassendateien, unterteilt. Außerdem sollte das erarbeitete Konzept ermöglichen, bereits bestehende Dateisysteme einzubinden, um Schulen die Arbeit beim Einpflegen der bestehenden Dateibestände zu gut es geht zu erleichtern. Dies wurde mit dem Strategy-Pattern erreicht (Abschnitt 3.2). Jede Schule kann im Administrationsbereich des Frontends selbst entscheiden, in welcher Art des Buckets die Dateien gelegt werden können. Dies kann eine AWS S3-Instanz, ein ownCloud-Server oder ein FTP-Zugriff sein. Wichtig ist nur, dass die nötige Strategie dafür implementiert ist. Dieses Konzept ist noch nicht vollständig im bestehenden System implementiert. Für die Pilotphase wird vom Projektteam eine geteilte S3 - Instanz bereitgestellt. Die URL sowie ein Import für den Schul-Bucket sollte in Zukunft noch einstellbar sein. Dies kann ebenfalls im Schul-Modell eingebaut werden. Außerdem gilt es eine Lösung für den Fall zu finden, dass eine Schule ein bestehenden Bucket benutzen möchte, der nicht dem Grundaufbau entspricht. Ein Lösung wäre hier eine Zwischenablage, so dass der Administrator die Dateien richtig verteilen kann. Eine weitere Anforderung war es, Dateien in den Unterricht und bei Hausaufgaben einbauen zu können. Diese Möglichkeit wurde im Abschnitt 4.2.2 mittels ckEditor gezeigt. Das gewählte Konzept erfüllt also alle genannten Anforderungen.

Im Verlaufe des Bachelorprojekts wurde das Konzept der Dateiverwaltung oft überdacht. Beispielsweise war die *path* - Variable, die in vielen Funktionen benutzt wird, ehemals in zwei Variablen aufgeteilt. Es gab einmal den Kontext-Pfad und den Dateinamen. Das Zusammenfassen bzw. Konkatenieren hatte den Vorteil, dass Funktionsdefinitionen übersichtlicher und die Benutzung einfacher wurde. Es wurde auch über die Umstellung auf eine flache Dateistruktur nachgedacht. Dies würde bedeuten, dass alle Dateien innerhalb eines Buckets

in einem einzigen Ordner untergebracht sind. Die Aufteilung in verschiedene Kontexte würde ein Feathers Proxy-Service im Schul-Cloud-Server übernehmen, welcher alle Daten zu den Dateien verwaltet.. Dieser würde sich außerdem um Berechtigungen kümmern. Viele Funktionen könnten vom File Storage Provider abstrahiert werden. Zum Beispiel müsste man beim Verschieben einer Datei nur den Datenbankeintrag ändern. Die Vorteile sind klar ersichtlich, man hätte im Client einen geringeren Aufwand bei der Ermittlung der richtigen Kontext-Pfade. Sehr viel Arbeit würde in den Server wandern. Man könnte dafür den bereits bestehenden *FilePermissionService* erweitern. Hier liegt aber auch ein Nachteil. Der Server würde ein Großteil der Arbeit erledigen. Man hätte zwar immer noch ein Strategy-Pattern, jedoch würde man viele Funktionen selbst implementieren müssen. Das Ziel der Arbeit war es zudem, auf bestehende Systeme zurück zu greifen zu können, um sich Arbeit zu sparen. Die flache Dateistruktur scheint viele Vorteile zu haben, jedoch hat man momentan ein funktionierendes Konzept für die Pilotphase und es bleibt zu evaluieren, wie dieses von den Testern aufgenommen wird. Wenn es zu Problemen kommt, sollte man über diese Alternative nachdenken.

Nach Beendigung des Bachelorprojekts waren noch längst nicht alle Aufgaben abgearbeitet. Dank der bereits laufenden Pilotphase sind viele interessante Wünsche und Anforderungen dazugekommen, welche es sich in Zukunft zu implementieren lohnt. Beispielsweise sollten die Zugriffsrechte erweitert werden. Alle für einen Nutzer geteilten Dateien sollen über einen zusätzlichen Menüpunkt zugreifbar sein. Außerdem wird ein Synchronisations-Client für den Desktop-Einsatz gewünscht. Ähnlich wie bei ownCloud soll es möglich sein, die Schul-Cloud Dateien auf der lokalen Dateiablage des genutzten Betriebssystems zu verwalten. Außerdem ist ein Service für die Vorschau von Dateien in Planung, der aber außerhalb des Bachelorprojekts entwickelt wird. Zur Verbesserung der Nutzerfreundlichkeit sollten Funktionalitäten wie das Suchen und Sortieren von Dateien implementiert werden. Ein erweiterte Admin-Ansicht auf die Dateiverwaltung wäre ebenfalls wünschenswert, um etwa Berechtigungen zu verwalten. Wichtiger Punkt ist außerdem, Dateien in den sogenannten *Lern-Store* [9] der Schul-Cloud einzubinden. Sehr engagierte Lehrer sollen die Möglichkeit haben, aufbereitetes Lernmaterial anderen Lehrern zugänglich machen.

6. Zusammenfassung

Eines der übergeordneten Ziele der deutschen Schul-Cloud ist es, Lernmaterial für alle und überall zugänglich zu machen. Diese Arbeit wollte erreichen, eine flexible Dateiverwaltung für dieses Vorhaben zu entwerfen und zu implementieren. Die Anforderungen wurden zu Beginn des Projekts klar definiert, so dass das Konzept auf diesen aufbauen konnte.

Es wurde sich für einen in Schulen aufgeteiltes Bucket-System entschieden, um Dateien logisch zu trennen. Diese werden in die einzelnen Kontexte des schulischen Alltags untergliedert, d.h. Kurse bzw. Fächer sowie Klassen und einzelne Individuen. Das Strategy-Pattern schaffte zudem die Möglichkeit, Buckets auf mehrere Server zu verteilen. Das Ziel einer grundlegend verteilten Dateiverwaltung wurde somit erfüllt. Viele Konzepte für die Dateiverwaltung wurden vom AWS S3-Standard abgeleitet. Da viele anderen Systeme wie zum Beispiel Microsoft Azure [34] auf den Standard aufbauen, liegt es nahe diesen im Schul-Cloud System zu benutzen. Das Strategy-Pattern ermöglicht es trotzdem, von diesem abzuweichen und ganz eigene Strategien zu implementieren.

Der Open-Source Status der Schul-Cloud soll hierbei helfen, zusammen mit Schülern und Lehrern die Schul-Cloud weiter zu verbessern. Strategien können nach und nach dazu implementiert werden und ins bestehende System eingepflegt werden. Eine Schule hat so zum Beispiel die Möglichkeit, für ihren Schul-Server ein eigene Anbindung zu schreiben, ohne sich zu stark an den Schul-Cloud Standard zu richten. Abschließend lässt sich sagen, dass die entworfenen Konzepte in der derzeit laufenden Pilotphase getestet und Feedback gewonnen werden müssen.

[34] Microsoft Azure - `https://azure.microsoft.com/de-de/`

Literatur

[1] *Schul-Cloud API Dokumentation*.
https://schulcloud.org:8080/docs/.

[2] ANDREAS BREITER, BJÖRN ERIC STOLPMANN, ANJA ZEISING: *Szenarien lernförderlicher IT-Infrastrukturen in Schulen*, 2015.

[3] BITKOM BUNDESVERBAND INFORMATIONSWIRTSCHAFT, TELEKOMMUNIKATION UND NEUE MEDIEN E.V.: *Studie „Bildung 2.0 - Digitale Medien in Schulen“*.
https://www.bitkom.org/noindex/Publikationen/2010/Studie/Studie-Bildung-2-0-Digitale-Medien-in-Schulen/BITKOM-Studie-Bildung-20.pdf.

[4] CHRISTOPH MEINEL, JAN RENZ, CATRINA GRELLA NILS KARN CHRISTIANE HAGEDORN: *Die Cloud für Schulen in Deutschland: Konzept und Pilotierung der Schul-Cloud*, March 2017.

[5] CLARKE, DAMIAN: *Conceptual Architecture Design and Configuration of a Peer to Peer File System for Schools and Medium Sized Business*, 2011.

[6] ITSLEARNING, GMBH: *Mythen und Fakten - Datenschutz bei Lernplattformen*.
http://info.itslearning.net/rs/655-PLS-373/images/mythen-und-fakten-datenschutz-bei-lernplattformen.pdf.

[7] KIEFER, NIKLAS: *Ergebnisse Umfrage 'Dateiorganisation in der Schule'*.
http://niklaskiefer.de/app/resources/umfrageergebnisse_dateien.xlsx.

[8] KIEFER, NIKLAS: *Umfrage 'Dateiorganisation in der Schule'*.
https://goo.gl/forms/xn4qcuC57jAQy0hq2.

[9] RENZ, JAN: *10 Forderungen an eine deutsche Schul-Cloud*.
https://blog.schulcloud.org/10-forderungen-an-eine-deutsche-schul-cloud
November 2016.

A. Anhang

Listing 1: Abstract file storage strategy

```
class AbstractFileStorageStrategy {
  constructor() {
    if (new.target === AbstractFileStorageStrategy) {
      throw new TypeError("Cannot construct AbstractFileStorageStrategy instances
          directly.");
    }
  }

  create() {
    throw new TypeError("create method has to be implemented.");
  }

  getFiles() {
    throw new TypeError("getFiles method has to be implemented.");
  }

  deleteFile() {
    throw new TypeError("deleteFile method has to be implemented.");
  }

  generateSignedUrl() {
    throw new TypeError("generateSignedUrl method has to be implemented.");
  }

  createDirectory() {
    throw new TypeError("createDirectory method has to be implemented.");
  }

  deleteDirectory() {
    throw new TypeError("deleteDirectory method has to be implemented.");
  }
}

module.exports = AbstractFileStorageStrategy;
```

Listing 2: Registrieren des FileStorage Services

```
module.exports = function () {
  const app = this;

  // Registriert die drei Services
  app.use('/fileStorage/directories', new DirectoryService());
  app.use('/fileStorage/signedUrl', new SignedUrlService());
  app.use('/fileStorage', new FileStorageService());

  // Holt die registrierten Services, um die Hooks zu initialisieren
  const fileStorageService = app.service('/fileStorage');
  const signedUrlService = app.service('/fileStorage/signedUrl');
  const directoryService = app.service('/fileStorage/directories');

  // Initialisiere "Before"-Hooks
  fileStorageService.before(hooks.before);
  signedUrlService.before(hooks.before);
  directoryService.before(hooks.before);

  // Initialisiere "After"-Hooks
  fileStorageService.after(hooks.after);
  signedUrlService.after(hooks.after);
  directoryService.after(hooks.after);
};
```

Listing 3: deleteFile() Funktion der AWS S3-Strategy

```
deleteFile(userId, path) {
  if (!userId || !path) return Promise.reject(new errors.BadRequest('Missing
      parameters'));

  // prüfe Berechtigung für die Löschen-Aktion
  return filePermissionHelper.checkPermissions(userId, path, ['can-write'])

    // finde gegeben Nutzer in Datenbank
    .then(res => UserModel.findById(userId).exec())
      .then(result => {
        if (!result || !result.schoolId) return Promise.reject(errors.NotFound(
            "User not found"));

```

```
12          // erstelle AWS-Verbindungsobjekts
13          const awsObject = createAWSObject(result.schoolId);
14
15          // stelle Parameter für das Löschen der Datei ein
16          const params = {
17            Bucket: awsObject.bucket,
18            Delete: {
19              Objects: [
20                {
21                  Key: path
22                }
23              ],
24              Quiet: true
25            }
26          };
27
28          // Führe Aktion aus
29          return promisify(awsObject.s3.deleteObjects, awsObject.s3)(params);
30    });
31  }
```

Listing 4: checkNormalPermissions() Funktion des filePermissionHelper

```
1   checkNormalPermissions(userId, filePath) {
2     let values = filePath.split("/");
3     if (values[0] === '') values = values.slice(1);
4     if (values.length < 2) return Promise.reject(new errors.BadRequest("Path is
          invalid"));
5     const contextType = values[0];
6     const contextId = values[1];
7     switch (contextType) {
8       case 'users':
9
10        // überprüfe, ob zu Prüfender Besitzer der Datei ist
11        if (contextId !== userId.toString()) {
12          return Promise.reject(new errors.Forbidden("You don't have permissions!
              "));
13        } else {
14          return Promise.resolve();
15        }
```

```
16
17        case 'courses':
18
19        // überprüfe, a) ob der Nutzer zum Kurs gehört, b) ob der Kurs überhaupt
              existiert
20        return CourseModel.find({
21          $and: [
22            {$or: [{userIds: userId}, {teacherIds: userId}]},
23            {_id: contextId}
24          ]
25        }).exec().then(res => {
26          if (!res || res.length <= 0) {
27            return Promise.reject(new errors.Forbidden("You don't have permissions!
                  "));
28          }
29
30          return Promise.resolve(res);
31        });
32
33      case 'classes':
34
35        // überprüfe, a) ob der Nutzer zur Klasse gehört, b) ob die Klasse ü
              berhaupt existiert
36        return ClassModel.find({
37          $and: [
38            {$or: [{userIds: userId}, {teacherIds: userId}]},
39            {_id: contextId}
40          ]
41        }).exec().then(res => {
42          if (!res || res.length <= 0) {
43            return Promise.reject(new errors.Forbidden("You don't have permissions!
                  "));
44          }
45
46        return Promise.resolve(res);
47      });
48
49      default:
50        return Promise.reject(new errors.BadRequest("Path is invalid"));
51      }
52    }
```

Listing 5: checkExtraPermissions() Funktion des filePermissionHelper

```
1  checkExtraPermissions(userId, fileKey, permissionTypes) {
2    if (!fileKey || fileKey === '') return Promise.reject(new errors.Forbidden("
         You don't have permissions!"));
3
4    return FilePermissionModel.find({key: fileKey}).exec().then(res => {
5
6    // Datenbankobjekt sollte einzigartig für Datei sein, trotzdem werden mehrere
          Einträge hier zusammengefasst
7    let permissions = _.flatten(res.map(filePermissions =>
         filePermissions.permissions));
8
9    // versuche Datenbankeintrag für bestimmten Nutzer zu finden
10   let permissionExists = permissions.filter(p => {
11     return JSON.stringify(userId) === JSON.stringify(p.userId) && _.difference(
           permissionTypes, p.permissions).length === 0;
12   }).length > 0;
13
14   if (!permissionExists) return Promise.reject(new errors.Forbidden("You don't
         have permissions!"));
15
16   return Promise.resolve();
17   }).catch(err => {
18     return Promise.reject(new errors.Forbidden("You don't have permissions!"));
19   });
20 }
```

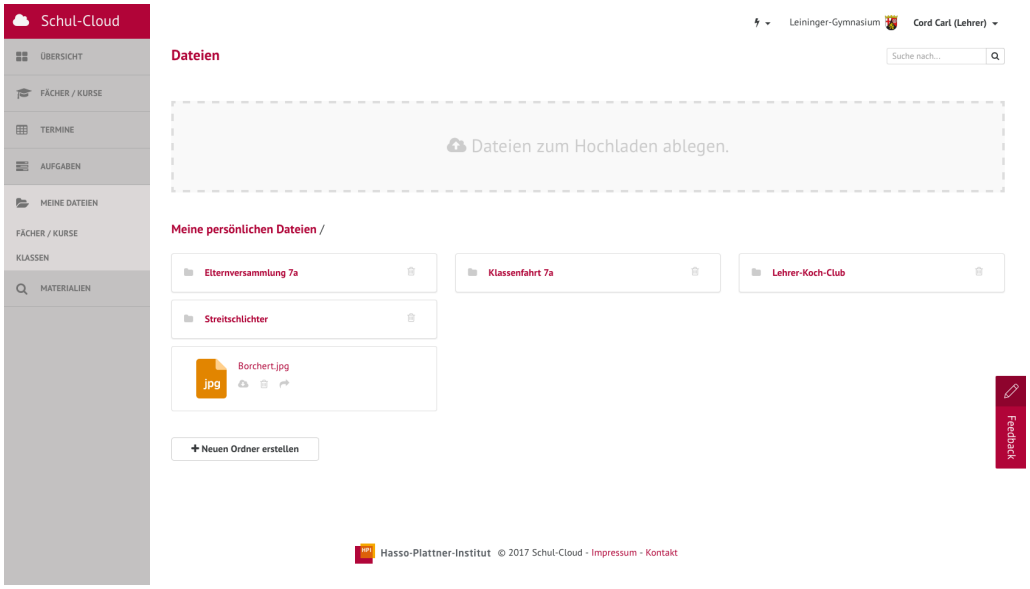

Abbildung 9: Ansicht für die persönlichen Dateien

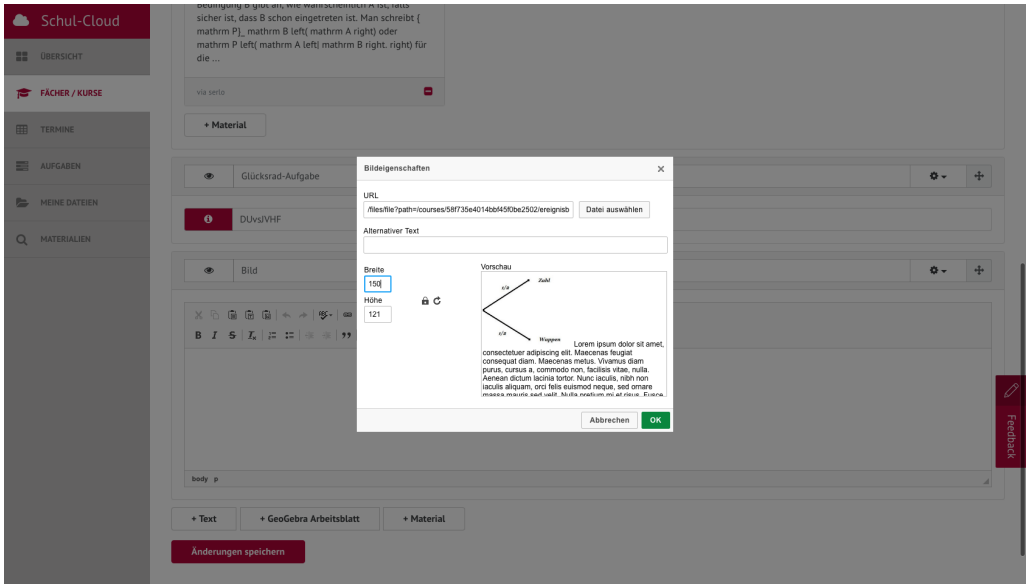

Abbildung 10: Ansicht für das Hinzufügen einer Datei in den Unterrichtskontext

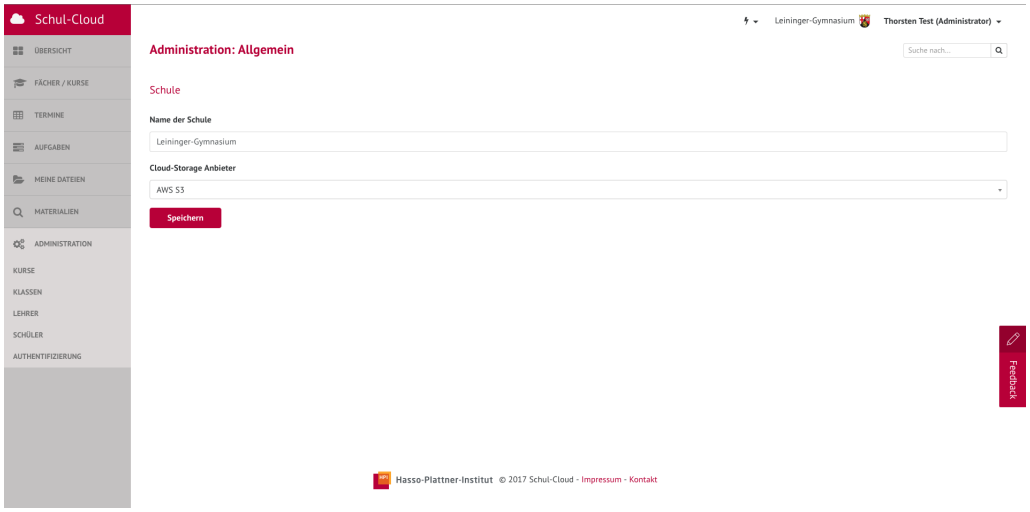

Abbildung 11: Ansicht für die Auswahl des File Storage Providers

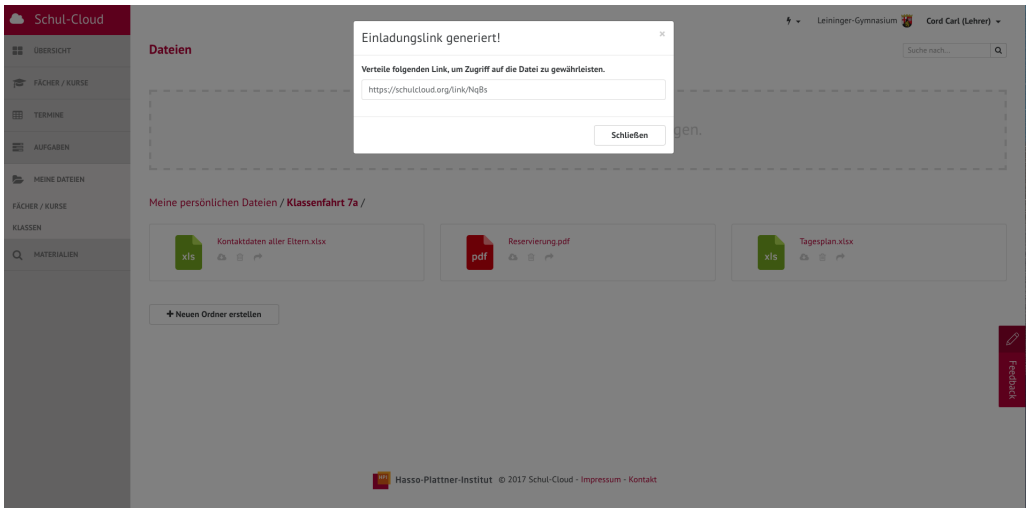

Abbildung 12: Ansicht für das Erstellen eines Zugriffslinks