3 Technische Dokumentation

Während des Projektseminars wurde für mehrere zu unterscheidende Bereiche programmiert: Windows 8, iOS, Android, Windows Phone 8 und ownCloud. Die technischen Aspekte der Implementierungen sollen im Folgenden näher erläutert werden.

3.1 Architektur

Während der Konzeptionierungsphase (siehe Kapitel 2.1) stellte sich die Flexibilität bezüglich Frontend und Backend der zu entwickelnden Lösung als wesentliche Anforderung heraus. Ziel des Projektseminars war es einerseits, mehrere Apps für verschiedene Endgeräte zu entwickeln, die Zugriff auf dieselbe Datengrundlage bieten. Umgekehrt sollte es ebenso möglich sein, durch einen hybriden Ansatz auf verschiedene Cloud-Speicherdienste über dieselbe Oberfläche zuzugreifen. Beides sollte durch die zu konzipierende Architektur unterstützt werden.

Als Cloud-Speicherlösung basieren die entwickelten Apps auf einer klassischen Client-Server-Architektur. Für die parallele Entwicklung mehrerer Apps in mehreren Teams waren Modularität und Kapselung daher wesentliche Kriterien. So konnten unabhängig von anderen Entwicklern und der zugrundeliegenden Server-Infrastruktur, von Anfang an verschiedene Teile der Apps verteilt entwickelt werden.

Außerdem ist die Wiederverwendbarkeit von Programmcode ein Indiz für die Qualität von Software. Um in der beschränkten Zeit eine maximale Produktivität zu erzielen, wurde daher eine einheitliche Architektur über alle Apps hinweg entwickelt, die gleichzeitig eine hohe Flexibilität wie auch die Nutzung von allgemeinen Funktionen für mehrere Entwicklungslinien gewährleistet. Als Grundlage dafür wurde die Entscheidung gefällt, sämtliche Apps auf Basis der Webtechnologien HTML5/CSS3 und JavaScript zu entwickeln.

Die Architektur selbst besteht aus 3 großen Schichten:

1. Der eigentliche Server, auf dem die Daten liegen
2. Die Logik-Schicht, die allgemeine Funktionen als Bibliothek bereitstellt
3. Die Anwendungs-Schicht zur Anzeige und Interaktion mit dem Benutzer

3.1.1 Die Server-Schicht

Der Server verwaltet die gesamten Daten. Für den Prototypen werden ownCloud (Version 5.0.6) und SharePoint 2013 als mögliche Server-Backends für die Apps vorgesehen. Die Serverseite selbst wird dabei nicht verändert, um beliebige Server unterstützen zu können. Jedoch besteht die Möglichkeit, weitere Funktionalitäten über Plugins in die jeweiligen Systeme einzubinden, wie es während des Projektseminars mit verschiedenen ownCloud-Plugins durchgeführt wurde (siehe Kapitel 3.6).

3.1.2 Die Logik-Schicht

Zwischen Anwendungsschicht und Server befindet sich eine Logikschicht, die als Bibliothek in jede App eingebunden ist. Sie ist für die Kommunikation mit dem Server zuständig und stellte weitere allgemeine Funktionen für die Apps bereit (siehe Abbildung 1: Projektplan 8).

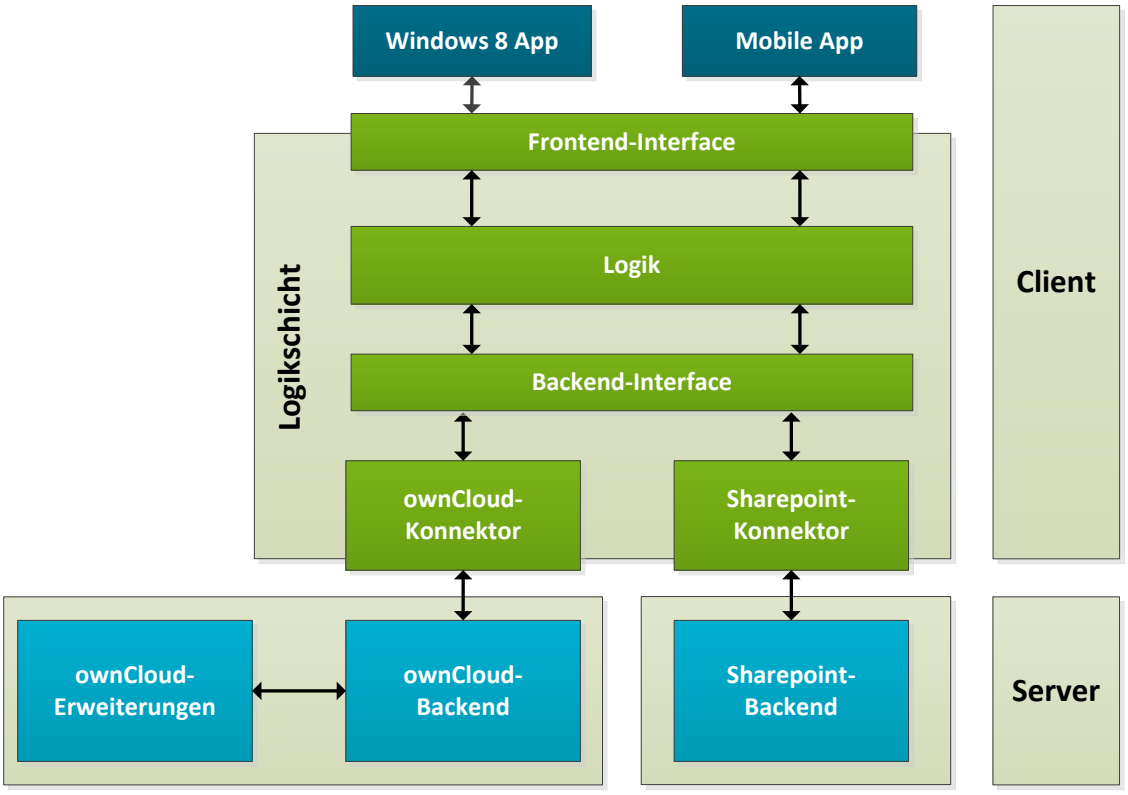

Abbildung 8: Allgemeiner Aufbau der Architektur

Die Logikschicht besteht aus mehreren Komponenten, die eine wesentliche Rolle für die Modularität der Architektur spielen. Hervorzuheben ist, dass sie nach oben und unten über vordefinierte Interfaces verfügt. Dadurch ist es möglich, mit Stümpfen oder

Dummy-Implementierungen zu arbeiten und für die App-Entwicklung entweder vom zugrundeliegenden Server oder gar der gesamten Logik zu abstrahieren. Die Anwendung lässt sich so mittels vordefinierter Rückgaben auf die korrekte Funktionalität testen. Außerdem führt dies zu einer Kapselung der verschiedenen Entwicklungsbereiche, sodass parallel an verschiedenen Teilen entwickelt werden kann, ohne dass sich die Zwischenstände gegenseitig beeinflussen.

Da JavaScript keine klassenbasierte Sprache ist, ist dort das Konzept eines Interfaces im Sinne von Programmiersprachen wie zum Beispiel Java nicht vorhanden. Daher bestehen diese Interface-Deklarationen aus einer Aufzählung von Funktionen. Beim Start der Applikation werden die Interfaces dann auf vollständige Implementierung der definierten Funktionen geprüft.

Besonderen Wert wurde bei der Spezifikation auf Einheitlichkeit und Übersichtlichkeit gelegt. Unter Anderem erwarten die definierten Funktionen deswegen höchstens drei Parameter. Der erste Parameter ist dabei ein JavaScript-Objekt, welches sämtliche erforderlichen und optionalen Parameter enthält. Die weiteren Parameter sind Callback-Funktionen für den Erfolgs- und den Fehlerfall: Da die Applikationen in hohem Maße von einer Onlineverbindung abhängig sind und die meisten Funktionen eine Kommunikation mit dem Server erfordern, wurde das Callback-Pattern als zentraler Bestandteil für die Funktionen ausgewählt. Dabei werden die Serveraufrufe asynchron getätigt, damit das Programm für die Zeit bis zur Antwort des Servers weiter ausgeführt werden kann. Die Rückgabe wird daraufhin analysiert und entweder der Fehler-Callback (wenn möglich mit genauerer Beschreibung des Fehlers) oder der Erfolgs-Callback (mit den gewünschten Daten) aufgerufen.

Bei der Implementierung der Funktionen ist außerdem zu beachten, dass diese allgemein verwendet werden können und nicht auf plattformspezifische Eigenheiten zurückgreifen. Dies ist zum Beispiel für Download und Upload von Dateien nicht gegeben, daher müssen diese beiden Funktionen im Rahmen der Initialisierung von der App mit übergeben werden.

Das Backend-Interface definiert eine Schnittstelle, um die verschiedenen Zugriffsmethoden auf die jeweiligen Server-Funktionen einheitlich ansprechen zu können. Für jedes unterstützte Backend existiert ein separater Konnektor, der dieses

Interface implementiert. Wesentliche Funktionen sind die Authentifizierung und Kommunikation mit dem Server, der Abruf der Daten sowie der Aufruf serverseitiger Funktionen. Zudem sind serverseitige Plugins auf Existenz zu prüfen, um weitere Funktionalitäten nur bei entsprechender Unterstützung anzuzeigen. Die genaue API-Spezifikation des Backend-Interfaces mit Parametern und Rückgaben ist in der Datei *backendInterface.js* dokumentiert (siehe Anhang A.b).

Das Frontend-Interface definiert die Schnittstelle zwischen der eigentlichen App und der Logikschicht. Während der Ausführung wird von der App eine Instanz des Frontends instanziiert, dies ist dann die einzige Verbindung zwischen der App und den weiteren Schichten. Es dient der Verwaltung des verwendeten Backends, der Erkennung von Fehlern beim Funktionsaufruf und leitet die Funktionen, welche eine Serverinteraktion benötigen, an das jeweilige Backend weiter (siehe Anhang A.a). Außerdem ist das Frontend verantwortlich für die Bereitstellung allgemeiner Funktionen, die unabhängig von der aktuellen Plattform allen Apps zur Verfügung stehen sollen. Diese sind in Module aufgeteilt, wie Abbildung 79 zu entnehmen ist.

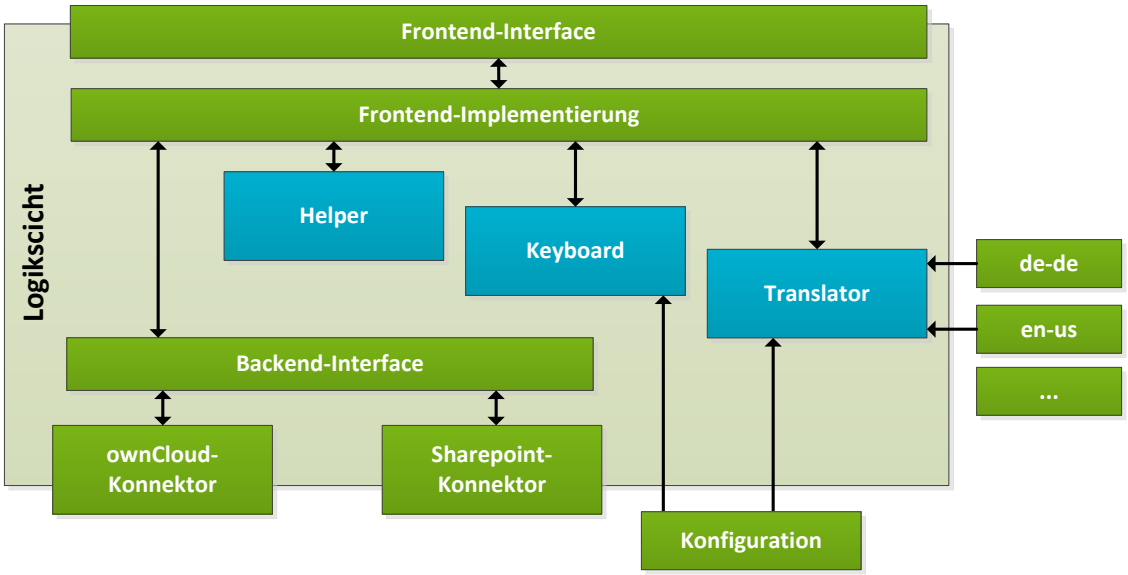

Abbildung 9: Detailaufbau der Logikschicht

3.1.2.1 Helper-Modul

Die Datei *helper.js* ist eine Sammlung allgemein nützlicher Funktionen, die der Bibliothek und den Apps zur Verfügung gestellt werden. Dazu gehören beispielsweise Sortierfunktionen, um Arrays von JavaScript-Objekten nach bestimmten Kriterien

sortieren zu können (*sortByParam*), Transformationsfunktionen, um Dateigrößen in verschiedenen Formen anzeigen zu können (*convertInAllFilesizes*) und Pfadangaben aufzubrechen (*convertPath)* und sonstige Hilfsfunktionen. Die genaue API ist in *helper.js* bzw. Anhang A.c dokumentiert.

3.1.2.2 Keyboard-Modul

Um die Apps über die Tastatur steuern zu können, erlaubt es dieses Modul, die Tastatur mit Funktionen zu belegen. Insbesondere werden auch Tastenkombinationen mit Strg-, Shift- und Alt-Taste unterstützt. Diese können jeweils mit der Funktion *bindKeystrokeEvent* hinzugefügt werden. Da die zur Verfügung stehenden Aktionen in der Regel abhängig vom aktuellen Zustand sind, in dem sich die App befindet, werden die Tastaturbefehle nach Möglichkeit in Kontexten organisiert.

Dazu werden Kontext und enthaltene Tastaturbefehle in der Konfigurationsdatei (*config.js*) als Parameter *keyboardContexts* hinterlegt. In der Anwendung kann dann über die Funktion *setKeystrokeContext* auf diese zugegriffen und diese mit den entsprechenden Funktionen verknüpft werden.

Das Modul verwaltet diese Kontexte und ändert bei einem Kontext-Wechsel die jeweiligen Eventhandler. Zur genauen Steuerung der Tastatur-Aktionen (Reichweite, HTML-Zielelemente, Art des Tastendrucks, etc.) existieren verschiedene Parameter, die in der Funktion *bindKeystrokeEvent* spezifiziert sind (siehe auch Anhang A.d) und in der Konfigurationsdatei mit übergeben werden können. Da viele Zielelemente der Tastaturbefehle üblicherweise auch per Mausklick aufgerufen werden sollen, ist es als Vereinfachung außerdem möglich, die Clickhandler ebenfalls über dieses Modul anlegen zu lassen.

Zum Einsatz kommt dieses Modul hauptsächlich in der Windows 8 Applikation, die Nutzung wäre aber auch für andere Frontends denkbar. Zum Wechseln des Tastaturkontextes wird der Befehl „*cloud.setKeystrokeContext({ context: "contextname", actions: {….}}“* aufgerufen. Das Array *actions* beinhaltet dabei die Zuordnung der in der config.js verwendeten *actions*-Parameter zu den jeweiligen Methoden. Um zu dem vorherigen Tastaturkontext zurückzukehren wird die Funktion *getPreviousKeystrokeContext* aufgerufen. Dies ist nötig, wenn für ein Fenster wie zum Beispiel das Ordner-erstellen ein eigener Keyboard-Kontext aktiviert wird. Wenn dieses

Fenster geschlossen wird kann dann der vorherige Tastaturkontext wiederhergestellt werden.

3.1.2.3 Translator-Modul

Dieses Modul implementiert die Mehrsprachigkeit der Apps. In der Konfiguration kann im Voraus eine Standardsprache hinterlegt sein. Bei der Initialisierung kann zudem basierend auf Benutzereinstellungen oder lokaler Indikatoren eine Benutzersprache festgelegt werden.

Für jede Sprache wird eine eigene Sprachdatei benötigt, welche die zu verwendenden Textbausteine enthält. Statische HTML-Elemente wie Überschriften, Texte und Buttons können direkt in den jeweiligen HTML-Dateien mit einem „lang“-Attribut versehen werden, das einen *translationKey* enthält. Bei Aufruf der *translateAll*-Funktion wird das aktuelle Dokument nach Elementen mit diesen Attributen durchsucht und mit den Übersetzungen ersetzt. Ist kein Eintrag für die aktuelle Sprache vorhanden, so wird die englische Übersetzung verwendet. Ist der *translationKey* vollständig undefiniert, so resultiert aus der Übersetzung „no such field“.

Über die *translate*-Funktion lassen sich einzelne Teile, zum Beispiel bei der Verarbeitung und Ausgabe in Skripten, direkt übersetzen. Dies kann zum Beispiel für dynamisch eingeblendete Elemente wie Benachrichtigungen genutzt werden.

Außerdem stehen eine *formatDate*- und *formatNumber*-Funktion zur Verfügung, um Daten und Zahlen ebenfalls den Eigenheiten der Benutzersprache anzupassen (siehe auch Anhang A.e).

3.1.2.4 Plugins

Zusätzlich zu den genannten Modulen wurde für die Bibliothek selbst auf mehrere Plugins von Fremdanbietern zurückgegriffen, welche nützliche Zusatzfunktionen bereitstellen und bei Bedarf auch von den App-Frontends genutzt werden können:

- **jQuery (Version 2.0.0):** http://jquery.com/

jQuery ist eine leistungsstarke JavaScript-Bibliothek, welche insbesondere die Arbeit mit DOM-Elementen erheblich vereinfacht und in vielen Punkten von plattformspezifischen Eigenheiten abstrahiert.

- **jQuery-dateFormat (Version 1.0):** https://github.com/phstc/jquery-dateFormat

  Dies ist eine Erweiterung zu jQuery, um Datumsangaben flexibel darstellen zu können. Die hinterlegten Textbausteine für die Rückgabe von relativen Datumsangaben (gestern,…) wurden angepasst, um die Mehrsprachigkeit der App unterstützen zu können. Diese sind nun ebenfalls in den App-Sprachdateien hinterlegt.

- **jQuery-scrollTo (1.4.3.1):**http://flesler.blogspot.de/2007/10/jqueryscrollto.html

  Ebenfalls als Erweiterung zu jQuery, dient scrollTo der flüssigen Animation von Sprüngen zwischen Elementen.

- **base64.js:** https://github.com/mshang/base64-js

  Dieses Plugin erlaubt es, Daten als base64 zu kodieren. Dies wird oft für die Datenübertragung zu Servern benötigt.

- **webdav.js:** https://github.com/dom111/webdav-js

  Wenn der Server wie bei ownCloud über eine WebDAV-Schnittstelle bereitstellt, kann dieses Plugin die clientseitigen WebDAV-Funktionalitäten ausführen.

- **pdf.js (Version 0.8.2):** http://mozilla.github.io/pdf.js/

  Dieses Plugin erlaubt es, PDF-Dateien in ein HTML-Canvas-Element zu rendern und ersetzt somit einen externen Betrachter. Da die Generierung eingebetteter Schriftarten über dynamisch generierte CSS-FontFace-Attribute in Windows 8 zu App-Abstürzen führt, wurde dieses Feature zugunsten direkter Schriftart-Generierung auf dem Canvas-Element ausgeschaltet (Z.17853). Zudem führen manche Konturen zu abstürzen, sodass die Generierung in einen try-catch-Block gekapselt wurde (Z.25243).

- **Codemirror (Version 3.13):** http://codemirror.net/

  Das Codemirror-Plugin ist ein komfortabler HTML-Betrachter für Textdateien. Für Features wie Syntax-Highlighting stehen für viele Sprachen eigene Erweiterungen bereit.

Mit den jeweiligen Interface-Implementierungen ergibt sich somit die in Abbildung 8 10 dargestellte Ordnerstruktur der Architektur.

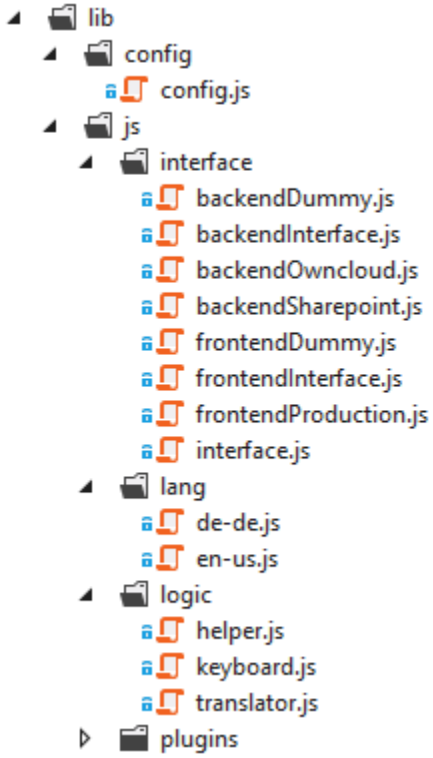

Abbildung 10: Ordnerstruktur der Bibliothek

3.1.3 Die Anwendungsschicht

Dies bezeichnet das jeweilige Frontend der App. Es ist verantwortlich für die Darstellung der Inhalte sowie die Interaktion mit dem Benutzer. Die Windows 8 App ist in Kapitel 3.2 näher erläutert, die mobilen Apps in Kapitel 3.3.

3.2 Windows 8 App

3.2.1 Grundlagen

Die App wurde unter Verwendung der Visual Studio 2012 Vorlage „Navigations-App“ erstellt. Diese Vorlage beinhaltet vordefinierte Steuerelemente zur Navigation zwischen einzelnen Unterseiten[1]. Die Basis dieser Vorlage sind die Dateien */default.html* und eine */js/default.js* sowie */js/navigation.js*, welche das Grundgerüst der gesamten Anwendung darstellt. Diese Dateien sind zu jedem Zeitpunkt im Hintergrund geladen, alle anderen Unterseiten werden dynamisch in diesen Container eingebunden.

- Die *default.html* bindet daher alle in der App benötigten JavaScript Erweiterungen und sonstige JavaScript Module ein.
- Die *default.js* initialisiert die App und berücksichtig dabei auch die unterschiedlichen Kontexte in der eine App gestartet werden kann (ActivationKind). Dazu zählen (Die letzten drei Kontexte betreffen in erster Linie das Verhalten der Unterseite für die Verzeichnisansicht. Daher werden diese im weiteren Verlauf beschrieben):
    - der komplette Neustart der App
    - die Fortsetzung der App aus dem Suspended-Zustand und die damit verbundene Wiederherstellung der Session-Einstellungen und Daten,
    - die Ausführung der App als Datenquelle für andere Apps (FileOpenPicker),
    - die Ausführung der App als Datenspeicher für andere Apps (FileSavePicker) und
    - die Ausführung der als Ziel für an die App geteilte Dateien (ShareTarget)

---

[1] Hinweis: Diese Steuerelemente wurden in der directoryView.js deaktiviert, da für die Ordnernavigation ein eigener Mechanismus implementiert wurde. Der dazu notwendige Code ist in der directoryView.js zu finden:

- WinJS.Navigation.history.backStack = [];
- $("header[role=banner] .win-backbutton").attr("disabled", "disabled");

- Die *navigatior.js* wurde fast unverändert von der Visual Studio 2012 Vorlage übernommen und nur um die automatische Übersetzung der Seite bei einer vollzogenen Navigation ergänzt.

Neben diesen Dateien wurde eine weitere zentrale Datei *appWindows.js* angelegt, welche sämtliche Funktionen beinhaltet, die global für die gesamte App verfügbar seien sollen. Diese Funktionen wurden im globalen Objekt „cloud.functions“ gespeichert und werden an mehreren Stellen in der App verwendet.

Das Navigationskonzept wurde genutzt, da die App aus zwei Unterseiten (/pages/…) bestand, welche in die default.html eingebunden werden:

- Eine Loginseite (*home.html*) die angezeigt wird, wenn der Benutzer nicht angemeldet ist oder ein automatischer Login mit gespeicherten Login-Daten beim Starten der App nicht möglich ist.

- Eine Verzeichnisansicht (*directoryView.html*), die nach dem erfolgreichen Login die Navigation zwischen den einzelnen Ordnern und das manipulieren von Dateien ermöglicht.

Eine Navigation zwischen der Login- und der Verzeichnisansicht wird über die Login und Logout Funktion der *appWindows.js* gesteuert. Neben diesen Seiten gibt es noch sogenannte Settings-Flyouts für die Einstellungen, die Dateihistorie und das Teilen sowie Freigeben von Inhalten. Diese sind im Verzeichnis *„/settings/…“* hinterlegt und bestehen jeweils aus einer .html Datei, einer .js und einer .css Datei. Diese Flyouts werden als Overlay über der aktuell angezeigten Seite dargestellt.

3.2.2 Die Loginseite

Die Loginseite besteht aus den Dateien *home.html*, *home.js* und *home.css* und wird aufgerufen, sofern der Benutzer sich ausloggt oder ein automatischer Login mit gespeicherten Logindaten beim Start der App nicht möglich ist.

Um zu vermeiden, dass die Loginseite bei dem Vorgang des automatischen Logins angezeigt werden muss, wurden die Funktionen zum Login bis auf die einzelnen Success- und Error-Handler in die *appWindows.js* ausgelagert. Sollte ein automatischer

Login bei dem Start der App möglich sein, navigiert diese direkt zur Verzeichnisansicht. Ansonsten wird als Error-Handler die Loginseite aufgerufen.

Die Loginseite selbst besteht aus zwei Spalten. In der ersten Spalte hat der Benutzer die Möglichkeit in einem ListView Element zwischen vorkonfigurierten Servern aus zu wählen oder aber die manuelle Konfiguration auszuwählen.

- **Vorkonfiguriert:** Die vorkonfigurierten Server werden in der Datei /*lib/config/config.js* definiert und anschließend in der Methode *initListView* der *home.js* in die ListView übertragen. Anschließend beinhaltet jede Kachel der ListView alle Informationen, die zum Verbinden mit den vorkonfigurierten Servern notwendig sind. Aktuell sind vorkonfigurierte Server für die ownCloud und die SharePoint 2013 Instanz der Universität Münster verfügbar.

- **Manuell:** Neben den vorkonfigurierten Servern, wird als letztes Element der ListView ein Element für die manuelle Konfiguration generiert. Sofern der Benutzer die manuelle Konfiguration auswählt, wird er nicht nur nach seinem Benutzernamen und seinem Passwort gefragt, sondern auch dazu aufgefordert, die notwendigen Serverinformationen einzugeben. Zu diesen Angaben zählen:

    - Der Servertyp: ownCloud oder SharePoint 2013
    - Der Serverpfad:
        - ownCloud z. B.: http://pscloud.uni-muenster.de/owncloud
        - SharePoint 2013 z. B.: https://pscloudservices.sharepoint.com
    - Der Pfad zur WebDAV-Schnittstelle (ownCloud) bzw. zur gewünschten Dokumentenbibliothek (SharePoint 2013)
        - ownCloud z. B.: /files/webdav.php
        - SharePoint 2013 z. B.: /DokumenteApp

Neben den für die Oberfläche wichtigen Operationen werden in der *home.js* auch die Logindaten für einen späteren Auto-Login gespeichert. Die Logindaten werden bei einem erfolgreichen Login (Methode: *loginSuccess*) in den persistenten und geräteübergreifenden App-Daten unter *Windows.Storage.ApplicationData.current.*

appData.roamingSettings gespeichert. Um die App an einem Autologin-Versuch zu hindern, sofern keine Daten hinterlegt sind, wird außerdem eine geräteübergreifende Variabel *loginStatus* gesetzt.

3.2.3 Die Verzeichnisansicht

Die Verzeichnisansicht (directoryView) wird dem Benutzer nach einem erfolgreichen Login angezeigt und ermöglicht es dem Benutzer in seinen Verzeichnissen zu navigieren und Dateien zu manipulieren. Die Verzeichnisansicht unterstützt verschiedene Darstellungskontexte, die teilweise in der *ready*-Funktion initialisiert/konfiguriert werden. Neben den Kontexten für den *FileOpenPicker*, den *FileSavePicker* und das *ShareTarget*, gibt es noch einen Kontext zum Verschieben von Dateien, den *fileMover*, und den normalen Kontext. In jedem dieser Kontexte ist die Navigation innerhalb der Ordner möglich. Wichtig zu erwähnen ist, dass die Aktionen, welche bei der Selektion oder Auswahl einer oder mehrerer Elemente durchgeführt werden, in der Funktion *initListView* initialisiert werden. Zu unterscheiden ist hier zwischen der normalen Auswahl eines Elementes mit der linken Maustaste (Einzelselektion = iteminvoked-Event) und der Selektion mit der rechten Maustaste (ggf. mehrfach Selektion = selectionchanged-Event). Dabei ist zu beachten, dass ein Linksklick sowohl das iteminvoked-Event, als auch das selectionchanged-Event ausführt. Die Funktion *initListView* registriert daher unter anderem je nach Kontext die jeweiligen Event-Handler.

3.2.3.1 Darstellungskontexte

Im Folgenden sollen nun die einzelnen Kontexte genauer betrachtet werden:

Normaler Kontext: Dieser Kontext stellt den Standard-Kontext der Verzeichnisansicht dar und verwendet die meisten der in der *directoryView.js* definierten Funktionen. Neben den Ordnern, werden in der Methode *reloadListView,* welche als Parameter das Array des Verzeichnisinhalts von der Backendmethode *cloud.getDirectoryContent* erwartet (Aufgerufen in der Methode *loadFolder*), auch die beinhalteten Dateien angezeigt. In diesem Modus kann der Benutzer in seinen Ordnern navigieren und mit den einzelnen Dateien interagieren (diese z. B. herunterladen, umbenennen, an andere Apps teilen). Darüber hinaus können gelöschte Dateien angezeigt werden (*displayDeleted*) oder aber Dateien verschoben werden (*moveObject).*

FileMover-Kontext: Beim Verschieben von Objekten wird die Variable *cloud.context.fileMover.isFileMover* auf true gesetzt. Diese Variable veranlasst die Verzeichnisansicht beim neu laden, sich im Kontext des *fileMovers* zu initialisieren. Dieser Kontext ermöglicht es dem Benutzer einen Ordner auszuwählen, in den er die zuvor ausgewählten Dateien (*cloud.context.fileMover.cutObjectPath[]* und *cloud.context.fileMover.cutObjectName[])* verschieben möchte. Daher werden ihm in diesem Kontext nur die vorhandenen Ordner zur Navigation angezeigt. Auch die Befehlsleiste (AppBar) zeigt in diesem Kontext nur die Buttons Verschieben und Abbrechen an. Andere Funktionen kann der Benutzer in diesem Kontext nicht aufrufen. Beim Verlassen des *FileMovers*, wird die Variable *fileMover.isFileMover* wieder auf false, die *cutObjects* auf null gesetzt und eine Navigation zur *directoryView.js* durchgeführt.

Hinweis: Zwar stellt das Laden und Verlassen des *fileMovers* eine Navigation dar, jedoch kann der Benutzer als Nebeneffekt des deaktivierten Back-Buttons diese Ansicht nur mit den dafür vorgesehenen Buttons erreichen und verlassen. Die Navigationselemente werden also nicht angezeigt.

ShareTarget-Kontext: Der Kontext *cloud.context.isShareTarget* ist jener, der aufgerufen wird, wenn andere Apps (Quell-Apps) Dateien an unsere App (Ziel-App) teilen. In diesem Fall wird ein Flyout unserer App am rechten Bildschirmrand angezeigt. Der *Kontext cloud.context.isShareTarget* ähnelt dem *fileMover* in seinen Eigenschaften. Statt den Buttons zum Verschieben und Abbrechen wird dem Benutzer jedoch nur ein Button für den Upload in den ausgewählten Ordner angeboten.

FileSavePicker-Kontext: Der Kontext *cloud.context.isSavePicker* ermöglicht es anderen Apps direkt Dateien in unserer App zu speichern und somit hochzuladen. Da Windows in diesem Kontext einen Speichern- und Abbrechen-Button automatisch bereitstellt, werden alle Buttons (abgesehen von den Navigationsbuttons) verborgen und deaktiviert. Um ein Event für den Speichern-Button zu definieren muss für das *fileOpenPicker*-Objekt (wird in der default.js beim Start der App in diesem Kontext in der Variable *cloud.context.pickerContext* gespeichert) das Event *targetfilerequested* registriert werden. Dieses Event bestimmt, welche Aktionen die Ziel-App mit der aus der Quell-App empfangenen Datei durchführen soll. Da es leider keine Möglichkeit gibt um herauszufinden wann die Quell-App die Datei vollständig bereitgestellt hat (im

Anschluss dessen der Upload durchgeführt werden kann), muss dies über einen Workaround gelöst werden. Die Funktion *onTargetFileRequested* erstellt eine temporäre Datei, die im Zugriffsbereich der Ziel-App liegt und übergibt diese zum Beschreiben an die Quell-App. Anschließend wird die Funktion *checkFileComplete* aufgerufen, die rekursiv darauf wartet, dass die Datei beschrieben wurde (Dateigröße > 0). Anschließend kann der Upload durchgeführt werden.

FileOpenPicker-Kontext: Der Kontext *cloud.context.isOpenPicker* stellt das Gegenteil des *FileSavePickers* da und ermöglicht es anderen Apps, Dateien direkt aus unserer App zu öffnen bzw. herunterzuladen. Anders als bei anderen Kontexten, werden in diesem neben den Ordnern auch die Dateien des aktuellen Verzeichnisses angezeigt. Wo eine Selektion einer Datei im normalen Kontext ggf. zu einer Dateivorschau führte, wird in diesem Kontext das selectionchanged -Event für die ListView in der Methode *initListView* mit der Funktion *addFileToBasket* belegt. Diese Funktion sorgt dafür, dass die ausgewählten Dateien temporär heruntergeladen werden und in einer Art Ablagekorb abgelegt werden. Bestätigt der Benutzer diese Auswahl mit dem Button „Öffnen“, werden die heruntergeladenen Dateien automatisch an die anfragende App übergeben. Von diesem Punkt an übernimmt die anfragende App wieder die Kontrolle.

3.2.3.2 Tastaturkontexte

Um die Bedienung mit der Tastatur zu erleichtern, wurden Tastenkürzel eingeführt, mit denen in der jeweiligen Ansicht bestimmte Befehle einfach ausgeführt werden können. Diese Tastenkürzel können in der Hilfe (z. B. über Alt + H erreichbar) eingesehen werden.

Technisch beruht die Tastaturbedienung auf dem Keyboard-Modul, welches sich um die Verwaltung der Tastaturbefehle kümmert und in Kapitel 3.1.2.2 beschrieben ist. Im Folgenden sollen ergänzend einige Hinweise zur Umsetzung in der Windows 8 App aufgeführt werden:

- Da die Kontexte zur Vereinfachung meist auch die Click-Event-Handler der einzelnen Buttons registrieren, dürfen diese in der *directoryView.js* nicht erneut mit Hilfe von *addEventListener* registriert werden.

- Der Kontext *directoryStart* stellt den Standardkontext dar. Es existiert ein Kontext „pdf“, der die gleichen Events und Shortcuts beinhaltet, im Grunde eine Kopie des Ganzen, und diese Events um die Tastaturbedienung während einer PDF Vorschau erweitert. Die Ursache hierfür ist, dass aktuell zu einem Zeitpunkt nur ein Kontext aktiv sein kann.

- Der in der Konfiguration angegebene *descriptionKey* wird genutzt, um im Hilfe-Flyout die Liste aller aktiven Tastaturbefehle zu generieren

- Um auch den click-Handler zu registrieren, muss das HTML-Element im *clickhandlerElement*-Parameter angegeben werden und der Parameter *addClickhandler* auf true gesetzt werden

3.2.3.3 Mehrsprachigkeit

Beim Start der App wird die Systemsprache über die Bibliotheks-Funktion *getSystemLanguage* ausgelesen und als Benutzersprache für die App hinterlegt.

Danach wird bei jeder Navigation zu einer neuen Seite die Funktion *translateApp* in *appWindows.js* aufgerufen. Diese nutzt die im Translator-Modul der Architektur beschriebene Unterstützung der Mehrsprachigkeit (siehe Kapitel 3.1). Zusätzlich wird als Windows-spezifische Eigenheit die Benennung der Charmbar-Flyouts übersetzt, da diese nicht über die normale Dokumentenstruktur vom Translator erreichbar sind.

Das Erweitern der App um weitere Sprachen ist einfach und kann folgendermaßen durchgeführt werden:

- Erstellung einer zusätzlichen Sprachdatei in *lib/js/lang/*

- Erweiterung der Funktion *getSystemLanguage* in */lib/js/interface/frontendProduction.js*, um die Unterstützung einer weiteren Sprache zu hinterlegen

- Erweiterung der Einstellungen in */settings/html/general.html* und */settings/html/general.js*, um zwischen den Sprachen wechseln zu können

3.2.3.4 Temporäre Downloads

Die Funktion *downloadFileTemporary* lädt eine Datei temporär herunter und speichert diese im temporären Verzeichnis der App. Diese Dateien können zu jedem Zeitpunkt nach dem Beenden der App vom Betriebssystem gelöscht werden. Ein temporärer Download wird an mehreren Stellen in der App benötigt. Dazu zählen das Öffnen von Dateien in einem externen Betrachter, das Teilen von Inhalten an andere Apps, der *fileOpenPicker* und die Dateivorschau. Damit die Datei nicht für jedes dieser Ereignisse erneut heruntergeladen werden muss, zum Beispiel der Benutzer öffnet erst die Vorschau eines Videos und möchte es dann jedoch in einem externen Betrachter öffnen, wird ein Verweis auf die heruntergeladene Datei im ListView-Item gespeichert. Das ausgewählte Element (*selectedItem*) wird markiert (*selectedItem.hasTemporaryFile*) und ein Verweis auf die Datei wird hinterlegt (*selectedItem.temporaryFile*). Wird erneut ein temporärer Download angefordert, so wird zuerst geprüft ob bereits eine temporäre Datei vorhanden ist. Erst wenn dies nicht der Fall ist, wird der Download durchgeführt. Da die ListView-Items bei jeder Navigation neugeladen werden, gehen die Verweise auf die bereits heruntergeladenen Dateien verloren und ein erneuter temporärer Download ist notwendig.

3.2.3.5 Dateivorschau

Sofern eine Datei mit einem Linksklick ausgewählt wird, wird unter anderem die Funktion *updatePreview* aufgerufen. Diese Funktion überprüft, ob für die ausgewählte Datei ein eine Vorschau generiert werden kann. Zuerst überprüft die Funktion ob das ausgewählte Element eine gelöschte Datei oder ein Ordner ist. In beiden Fällen wird keine Vorschau generiert. Anschließend wird überprüft, ob für den Dateityp des ausgewählten Elements ein Vorschautyp (*previewType*) in der *config.js* hinterlegt ist. Sofern dies der Fall ist kann eine Vorschau generiert werden.

Es gibt zwei Typen der Vorschau von Dateien:

1. Vorschau ohne vorheriges Herunterladen der Datei

2. Vorschau, die einen temporären Download benötigt

Der erste Typ einer Vorschau wird momentan nur bei .docx Word-Dokumenten verwendet. Der ownCloud Server konvertiert .docx-Dokumente in HTML-Code und

übergibt diesen an die App. Dieser Code kann dynamisch in den Vorschau-Container eingefügt werden.

Beim zweiten Typ der Vorschau ist ein vorheriger temporärer Download der Datei nötig. Anschließend sorgt die Funktion *setFilePreviewHTML* dafür, dass die letzte Dateivorschau ausgeblendet wird und eine neue Vorschau abhängig vom Dateityp generiert wird. Nach Möglichkeit werden zur Darstellung der Dateien die in HTML5 integrierten Viewer verwendet. Zusätzlich kommt das Plugin pdf.js zum Einsatz, um PDF-Dateien in einem HTML-Element rendern zu können sowie das Plugin Codemirror, welches die Darstellung von Textdateien unterstützt.

Aktuell kann die App die folgenden Formate darstellen: *.bmp, .c, .cbz, .cpp, .c#, .css, .docx, .diff, .gif, .gitattributes, .gitignore, .hs, .html, .java, .js, .jpeg, .jpg, .log, .lua, .mp3, .p, .pdf, .perl, .php, .png, .py, .r, .rb, .s, .sql, .stex, .tex, .txt, .vb, .vba, .vbw, .vbs, .xml, .mp4, .wmv, .m4v, .webm*

3.2.3.6 Teilen von Inhalten an andere Apps

Das Teilen von Inhalten an andere Apps ist eine zentrale von Windows 8 bereitgestellte Funktion, die über den Teilen-Button der Charm-Bar aufgerufen werden kann. Um das Teilen von Inhalten aus der App zu ermöglichen, muss im normalen Kontext ein Event für das *datarequested*-Ereignis des *DataTransferManagers* registriert werden:

```
var dataTransferManager = Windows.ApplicationModel.DataTransfer.
DataTransferManager.getForCurrentView();dataTransferManager.addEventListener(
"datarequested", this.dataRequested);
```

Das in der App definierte Event befindet sich in der Funktion *dataRequested*. Grundlegend wird diese Funktionalität jedoch über die Funktion des temporären Downloads von Dateien ausgeführt. Soll eine Datei an eine andere App geteilt werden, so wird zuerst geprüft ob bereits eine temporäre Datei für das aktuell angezeigte Verzeichnis vorhanden ist. Wenn dies der Fall ist wird kein erneuter Download durchgeführt und die Datei kann direkt an eine andere App geteilt werden. Eine eigene als Eventhandler zum Teilen von Inhalten ist ausschließlich notwendig um das *requested*-Objekt des Aufrufereignisses zu erlangen. Dieses Objekt wir dazu verwendet, um der Charm-Bar mitzuteilen, dass die Datei bereit zum Teilen ist. Sollte noch keine

temporäre Datei vorhanden sein, so muss die Datei zunächst heruntergeladen werden bevor sie an eine andere App geteilt werden kann. Die Charm-Bar wartet solange bis die heruntergeladene Datei an den Item Container des Teilen-Flyouts übergeben wurde (*shareRequest.data.setStorageItems([targetFile])*) und ihr mitgeteilt wurde, dass der Vorgang abgeschlossen ist und die Datei fertig zum Teilen ist (*deferral.complete*).

3.2.3.7 Gelöschte Dateien

Gelöschte Dateien können durch ein Button Event eingeblendet werden. Die Funktion *displayDeleted* sorgt in erster Linie dafür, dass der aktuelle Kontext der App erweitert wird, indem die Variable *cloud.context.showDeletedFiles* auf True gesetzt wird. Diese Variable sorgt sowohl dafür, dass in der Funktion *loadFolder* auch gelöschte Dateien hinzugezogen werden, als auch für die Einschränkungen der Aktionen, die mit einem ausgewählten gelöschten Element durchgeführt werden können. So werden gelöschte Dateien bei Aktionen, die mehrere Elemente gleichzeitig behandeln, zum Beispiel das Verschieben, mit einer Fehlermeldung ignoriert. Daneben werden außerdem in der Funktion *updateAppbar* alle Funktionen deaktiviert, die nicht mit gelöschten Dateien durchgeführt werden können. Die Wiederherstellen-Funktion nimmt in diesem Fall eine Sonderrolle ein und wird nicht deaktiviert.

3.2.3.8 Multi-Events

Einige Operationen können gleich mehrere Dateien auf einmal manipulieren. So ist es beispielsweise möglich mehrere Dateien gleichzeitig herunterzuladen, zu löschen, wiederherzustellen oder zu verschieben. Um dies zu ermöglichen wird meist eine Schleife über alle selektieren Elemente durchlaufen, die jeweils die Einzeloperationen durchführt. Zu diesem Zwecke wurden die Button-Events oft von den eigentlichen Methoden getrennt. Ein Beispiel dafür ist die Funktion *restoreFileButtonEvent*, welche jedes einzelne selektierte Element überprüft und gegebenenfalls über die Funktion *restoreFile* wiederherstellt. Ein anderes Beispiel ist der Download. Hier werden die herunterzuladenden Dateien jedoch einzeln in das Array der Downloadparameter übertragen und anschließend einmalig die Methode *downloadAndSaveFile* aufgerufen. Der Download der einzelnen Dateien wird dann anschließend getrennt in *appWindows.js* durchgeführt.

3.2.3.9 Settings-Flyouts

Einige Funktionen wurden mit sogenannten Flyouts umgesetzt. Zu diesen Funktionen zählen das Freigeben von Inhalten an andere Benutzer und das Wiederherstellen vergangener Dateiversionen. Beide Funktionen benötigen ein Serverseitiges Plugin. Stellt der Server diese nicht bereit, so werden die Buttons zum Öffnen der Flyouts nicht angezeigt.

Wird eine dieser beiden Funktionen aufgerufen, wird das selektierte Element in einer globalen Variabel (z. B. *cloud.context.history.file*) gespeichert und anschließend wird das jeweilige Flyout mit der Funktion *WinJS.UI.SettingsFlyout.showSettings* aufgerufen. Diese Flyouts sind, wie auch alle anderen Einstellungs-Flyouts, im Ordner „*/settings/...*“ hinterlegt.

Die Flyouts für das Freigeben von Inhalten und das Wiederherstellen von Dateiversionen orientieren sich an der Umsetzung der Verzeichnisansicht. Jedes Flyout verfügt über mindestens ein ListView Element, welches von der Logik identisch zum Verzeichnisinhalt befüllt wird.

Im Share-Flyout existieren zwei voneinander getrennte ListViews. Die erste ListView stellt die vorhandenen Benutzer dar, an die ein Objekt geteilt werden kann. Der Benutzer kann mit Hilfe eines Input Feldes nach den Benutzernamen suchen. Hat der Benutzer keine Eingabe gemacht, so wird ein Ausschnitt aller Benutzer angezeigt. Sollte der Benutzer keine Berechtigung zum Teilen des ausgewählten Elements hat, so kann weder ein Freigabelink noch die Liste der bereits vorhandenen Freigaben generiert werden. Da der Freigabelink als erstes generiert wird, wird das Share-Flyout mit der Fehlermeldung des Freigabelinks geschlossen. Eine Möglichkeit serverseitig anzufragen, ob der Benutzer das Recht dazu hat, das ausgewählte Element mit anderen zu teilen, existiert noch nicht.