Hochschule für Telekommunikation Leipzig (FH)
Profilierung Mobile Applikationen

Dokumentation EtherApp

Thema: Entwicklung einer App zur Verwaltung von Etherpad Lite

Vorgelegt von: Ferdinand Malcher
Martin Stoffers

Betreuer: Prof. Dr. Ulf Schemmert

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
UI	User Interface, Benutzeroberfläche
URL	Uniform Resource Locator

1 Systembeschreibung

1.1 Etherpad Lite

Etherpad Lite[1] ist ein kollaborativer Online-Editor. Anwender können Textdokumente, sogenannte Pads, gemeinschaftlich und in Echtzeit online bearbeiten. Die Oberfläche ist browserbasiert. Innerhalb der Pads ist durch farbliche Markierung gekennzeichnet, welche Bearbeitungen von welchem Nutzer vorgenommen wurden. Eine Instanz des Etherpad Lite ist standardmäßig offen. Damit kann jeder Anwender Pads anlegen und bearbeiten. Das System speichert Revisionen des Pad-Inhalts, die später abgerufen werden können. Die Oberfläche bietet außerdem die Möglichkeit zum Export des Inhalts in verschiedene Dokumentformate sowie einen Onlinechat zur Verständigung unter den Autoren.
Etherpad Lite ist Open-Source und damit quelloffen und frei.

1.2 Abgrenzung und Zielstellung

Die Oberfläche des Etherpad Lite ist schlicht und übersichtlich gehalten. Sie bietet benutzerrelevante Funktionen, die sich auf ein einzelnes Pad beziehen. Administrative Funktionen, die bestimmten Benutzerkreisen vorbehalten bleiben müssen, sind auf der Oberfläche nicht implementiert. Es gibt somit keine Möglichkeit, z.B. alle verfügbaren Pads anzuzeigen oder einzelne Pads zu löschen.
Es existiert eine umfangreiche HTTP-API, über die diverse Funktionen verfügbar gemacht werden. Für die praktische Verwendung dieser API ist ein Frontend sinnvoll.
Das Projekt EtherApp hat zum Ziel, einen Teil der API-Funktionen abzubilden und eine App zur Administration verschiedener Etherpad-Lite-Instanzen zu entwickeln. Das beinhaltet insbesondere folgende Funktionen:

- Anzeige aller Pads
- Anzeige von Meta-Informationen für einzelne Pads (Benutzer online, Revisionen, Datum, ...)
- Anlegen neuer Pads
- Löschen von Pads
- Anzeige des Pad-Inhalts
- Teilen der Pad-URL über soziale Dienste und E-Mail
- Rücksetzen des Inhalts auf ältere Revision
- Anzeige und Verwaltung von Gruppen
- Verwaltung mehrerer EPL-Instanzen und Profilverwaltung

2 Implementation

2.1 Zugriff auf die HTTP-API

Das EPL bietet eine umfangreiche HTTP-API[2] an. Sie stellt eine Reihe von Funktionen für administrative Aufgaben bereit, die über die offene Weboberfläche nicht zur Verfügung stehen. Der Zugriff auf die API ist nur mit einem Shared Secret (API Key) möglich, der vom Administrator festgelegt werden muss.
Die API gibt Antworten in JavaScript Object Notation (JSON) zurück. Damit lassen sich Arrays oder einfache Strings gleichermaßen per HTTP übertragen. Ein Beispiel für einen Rückgabewert aus der API ist in Listing 1 dargestellt (Methode `listAllPads`).

Listing 1: Rückgabewert der API zur Methode listAllPads

```
1 {"code":0,"message":"ok","data":{"padIDs":["interrail","
    ↪ interview","jabberstatusdienst","java","Java","java2"]}}
```

Für den Zugriff auf die API existieren bereits Java-Bibliotheken, die alle API-Funktionen auf Methoden einer Java-Klasse abbilden. Für das Projekt EtherApp kam dabei die Implementation[3] von Jordan Hollinger zum Einsatz.
Diese Bibliothek beinhaltet eine Klasse `EPLiteClient`, welche mit den Zugangsdaten für die EPL-API initialisiert wird (siehe auch Unterabschnitt 2.4). Mit einem Objekt dieser Client-Klasse können Anfragen auf die jeweilige API durchgeführt werden, die als Java-Datenstruktur (`String` oder `HashMap`) zurückgegeben werden.

2.2 Selbst definierte Listen

Das Android-SDK stellt mit dem `ListView` ein View-Element zur Verfügung, mit dem Elemente in einer scrollbaren Liste angezeigt werden können.
Zur Anpassung von Daten und das tatsächliche Darstellen in der Liste wird ein Adapter benötigt. Android stellt bereits eine Reihe von Adaptern zur Verfügung (`BaseAdapter`, `ArrayAdapter`, ...). Deren Nachteil ist jedoch eine festgelegte Darstellungsweise, denn auf einem Listenelement ist lediglich ein `TextView` dargestellt.
In unserem Anwendungsfall sollten in den Listenelementen der Padliste neben dem eigentlichen Pad-Namen zusätzliche Statusinformationen zum Pad sowie ein Löschen-Button angezeigt werden. Dieses Ziel konnte nur mit einem eigenen Adapter[1] und einem selbst gestalteten Listenelement[2] erreicht werden. Der entgültige Entwurf der Padliste ist in Abbildung 1 zu sehen.

2.3 Asynchrones Laden der Listenelemente

Um eine komplette Padliste inklusive der Metadaten zu jedem Pad abzurufen, sind folgende Anfragen an die HTTP-API nötig:

- Abrufen der Padliste (nur IDs)

[1] `de.etherapp.adapters.PadlistBaseAdapter`
[2] `de.etherapp.beans.PadlistItem`

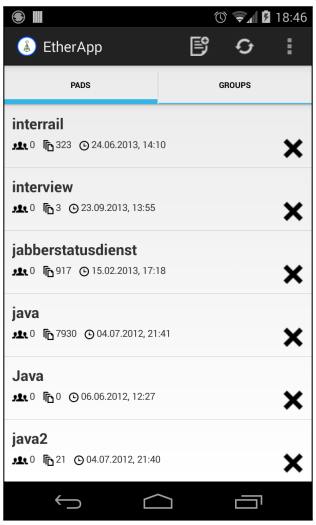

Abbildung 1: PadlistActivity mit mehreren PadItems

- Abrufen der Metadaten zu jedem einzelnen Pad
  - Anzahl der User online
  - Anzahl der Revisionen
  - Datum der letzten Bearbeitung

Für jedes Pad sind also drei HTTP-Anfragen abzusetzen. Bei einem Volumen von 300 Pads sind damit 901 HTTP-Anfragen nötig, um alle Daten der Padliste abzurufen, auch wenn der Benutzer diese zunächst gar nicht benötigt! Neben der dadurch ausgelösten Serverlast ist auch die resultierende Wartezeit für den Benutzer nicht akzeptabel. Dieser Umstand erfordert eine Lösung, mit der die Inhalte eines Listenelements erst dann geladen werden, wenn das Element auch angezeigt wird.
Dieses Ziel kann mit einem `AsyncTask` erreicht werden. Zunächst wird die Liste aller Pads abgerufen und das `ListView` mit Elementen gefüllt. Jedes angezeigte Listenelement startet dann einen neuen Thread (`AsyncTask`) für jeden zu ladenden Wert. Der Thread stellt asynchron eine Anfrage an die API und schreibt die Ergebnisse direkt in die Views des Listenelements. Das asynchrone Laden ist beim Scrollen der Liste an der Zeitverzögerung zu erkennen.

2.4 Verwaltung mehrerer EPL-Instanzen

Zur Verwaltung mehrerer Instanzen des EPL auf verschiedenen Servern wurde eine Profilverwaltung implementiert. Die benötigten Informationen für den Zugriff auf eine EPL-Instanz sind in Tabelle 1 zu sehen. In den Einstellungen können Daten für verschiedene APIs eingetragen werden. Jeweils eine API kann zur Verwendung ausgewählt werden. Beim Start der App wird die zuletzt ausgewählte API verwendet. Startet die App zum ersten Mal, wird der Benutzer direkt zum Anlegen einer neuen API aufgefordert.

Information	**Beschreibung**
API-Name	eindeutiger Name einer EPL-Instanz
URL	Adresse der API
Port	verwendeter Port
API-Key	geheimer Schlüssel für den Zugriff

Tabelle 1: Benötigte Informationen für eine EPL-Instanz

2.5 Persistenter Datenspeicher

Einige Parameter der EtherApp müssen persistent gespeichert werden, z.B. die vom Benutzer definierten APIs und Grundeinstellungen. Für diese Aufgabe gibt es zwei Lösungsansätze: Shared Preferences und eine SQLite-Datenbank.

Die **Shared Preferences** sind Schlüssel-Wert-Paare, auf die systemglobal von Applikationen zugegriffen werden kann. Da uns die Implementation einfacher erschien, sollte diese Methode zum Speichern der API-Informationen verwendet werden. Dazu wurde dem Schlüssel (z.B. `apikey`) ein Index angefügt (z.B. `apikey2`) um die Zugehörigkeit zu einer API-Einstellung, die aus mehreren Schlüsseln besteht, erkennbar zu machen. Diese Variante funktionierte, hatte aber ihre Grenzen. Da es keine Funktion gibt, alle existierenden Schlüssel abzurufen, musste immer die Anzahl der existierenden APIs mitgespeichert werden, um durch die Liste iterieren zu können. Sobald ein Eintrag gelöscht wurde, traten Konsistenzprobleme auf, die nur mit viel Aufwand wieder zu beheben gewesen wären. Wir entschieden uns daher für eine Migration auf **SQLite**, ein leichtgewichtiges relationales Datenbanksystem auf Dateibasis.
Android stellt bereits die nötigen Schnittstellen für SQLite-Datenbanken zur Verfügung. Die Daten lassen sich mit einer relationalen Datenbank wesentlich leichter auslesen und verändern, obgleich die Implementation und Einrichtung etwas aufwendiger ist, als für die Shared Preferences.
Zum Zugriff auf die Datenbank wurde die Klasse `de.etherapp.sql.DBHandler` als Extension der Elternklasse `SQLiteOpenHelper` angelegt. Sie stellt die Datenbank für den Schreib- oder Lesezugriff zur Verfügung und regelt die Initialisierung der Datenbank beim ersten Start der App oder beim Upgrade.

Für die EtherApp wurden zwei Tabellen angelegt. `ea_padapi` beinhaltet alle Parameter für die API-Definition. Zur Identifizierung eines Datensatzes wird eine interne UUID verwendet. Die Tabelle `ea_pref` nimmt Schlüssel-Wert-Paare für verschiedene Einstellungen auf, z.B. die UUID der zuletzt ausgewählten API.
Die Struktur der Datenbanktabellen ist nachfolgend dargestellt.

ea_padapi		
apiid	VARCHAR(64)	PRIMARY KEY
apiname	VARCHAR(255)	NOT NULL
apiurl	VARCHAR(255)	NOT NULL
port	INT	NOT NULL
apikey	VARCHAR(255)	NOT NULL
timestamp	UNIX_TIMESTAMP	

ea_pref		
name	VARCHAR(255)	PRIMARY KEY
value	VARCHAR(255)	NOT NULL

2.6 TabActivity

Zur Navigation zwischen den Haupt-Activities (Padliste, Gruppenliste, ...) entschieden wir uns für eine Tab-Ansicht. Am oberen Rand des Bildschirms sollte eine Tab-Leiste zu sehen sein, beim Anklicken sollte der Inhaltsbereich unten aktualisiert werden (siehe Abbildung 2).
Für diesen Zweck ist das View-Element `TabHost` entwickelt worden. Innerhalb einer `TabActivity` sorgt es für die Auslieferung von Tabs. Es stellt einen Rahmen zur Verfügung, in den je nach aktivem Tab eine Activity geladen wird.
Später fiel uns auf, dass dieses Feature als *deprecated* eingestuft ist und nicht mehr verwendet werden soll. Stattdessen sollen Tab-Ansichten mit sogen. Fragments implementiert werden. Die Migration auf eine Fragment-Umgebung hätte jedoch eine vollkommene Umstrukturierung der Software mit sich gebracht und wurde aus Zeitgründen nicht durchgeführt.

Abbildung 2: TabActivity mit Tabs und Inhaltsbereich

2.7 Datenstruktur

Alle Daten (APIs, Pads, Gruppen) der EtherApp werden in einer Klassenstruktur gespeichert. Die gesamte Struktur ist in Abbildung 3 dargestellt.
Die Wurzel der Struktur ist die statische Klasse `GlobalConfig`. Sie enthält eine `HashMap` mit allen angelegten APIs, jeweils Objekte der Klasse `PadAPI`. Innerhalb der API beinhaltet eine `HashMap` alle Pads inklusive ihrer Parameter (sofern schon abgerufen). Ein Pad ist in einem Objekt der Klasse `Pad` repräsentiert.
In der `GlobalConfig` sind außerdem der `DBHandler`, die `MainActivity` (Host für die Tab-Ansicht) und die aktuell ausgewählte API als Objekte hinterlegt.
Die Klasse `PadThread` ruft beim Start die komplette Liste aller Pads asynchron aus der API ab.

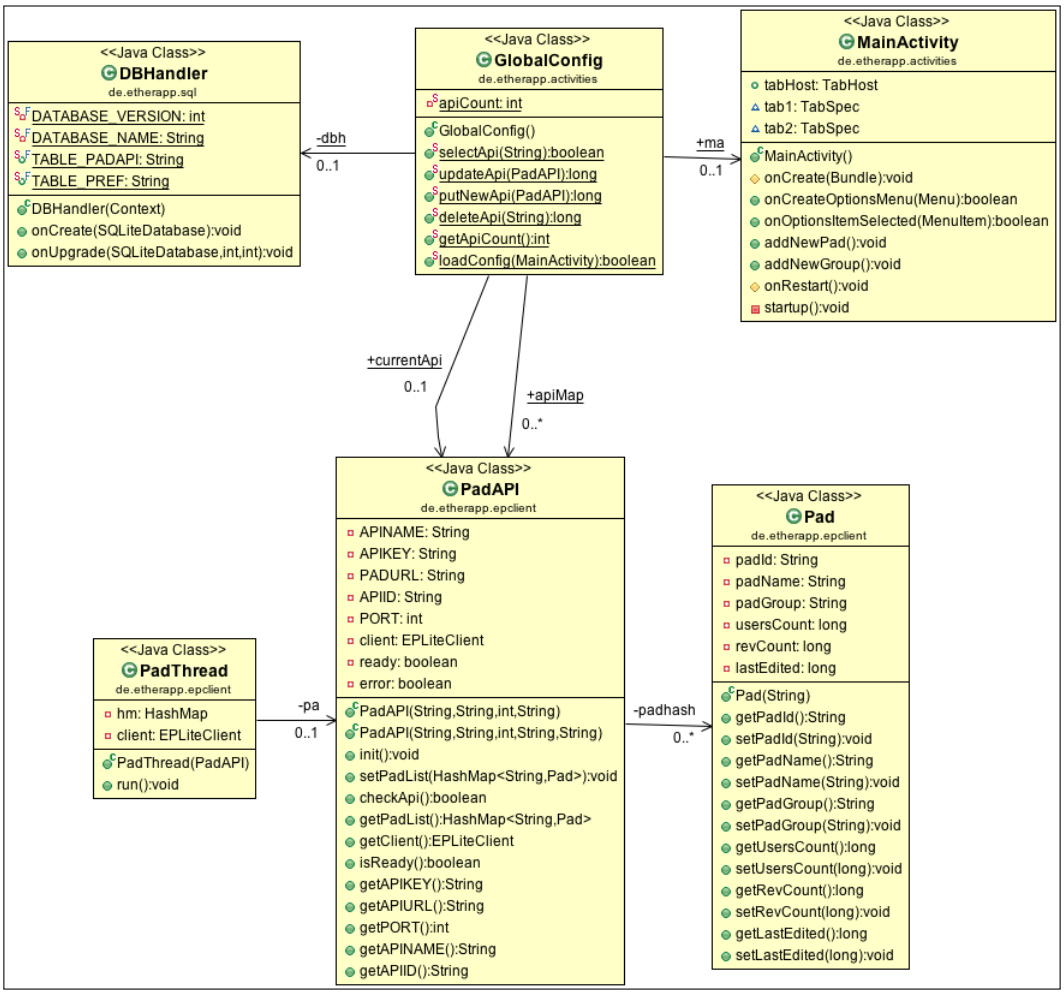

Abbildung 3: Klassendiagramm der EtherApp

3 Probleme bei der Umsetzung

Hinsichtlich des Software-Engineering-Prozesses wurde das Modell des Extreme Pair Programming mit Rapid Prototyping verfolgt. Die Implementation erfolgte zum großen Teil gemeinschaftlich, mit dem Ziel, einen schnellen Prototypen zu entwickeln. Dieses Vorgehen und der Umstand, dass wir uns das weite Feld der Software-Entwicklung auf einer Android-Plattform erst erschließen mussten, führten zu Unsicherheiten in der Umsetzung.

Die bestehenden Konzepte wurden mehrfach überarbeitet und an gewonnene Erkenntnisse angepasst. Viele Eigenschaften, die noch zu Beginn des Entwicklungsprozesses sinnvoll erschienen, erwiesen sich später als überflüssig.
Einige Problemstellungen sind nachfolgend kurz beschrieben.

3.1 Asynchrone Netzwerkkommunikation

Aus der Theorie ist bekannt, dass blockierende Netzwerkaufrufe nicht im Hauptthread der Anwendung durchgeführt werden sollten, da die gesamte Anwendung (und damit auch das UI) dann blockiert. An einigen Stellen (z.B. beim Starten der App und dem initialen Laden der Padliste) kann ein Benutzer eine Wartezeit allerdings in Kauf nehmen.
Der Einfachheit halber (und weil diese Eigenschaft die Bedienbarkeit der App nicht in hohem Maße einschränkt), sollten die meisten Netzwerkaktionen synchron im Hauptthread erfolgen. Android verbietet in der Default-Konfiguration jedoch Netzwerkzugriff im Hauptthread (`android.os.NetworkOnMainThreadException`).
Teilweise konnten wir die Netzwerkaktionen auf Threads auslagern (`PadThread` zum Abrufen der Padliste ohne Metadaten oder `Asynctask`s für Metadaten der Pads). Einfache Threads haben allerdings dann ihre Grenzen, wenn die Daten direkt in ein `View`-Element geschrieben werden sollen. Einzelne Versuche, die Funktionalität mit Services zu implementieren, scheiterten.
Daher entschieden wir uns schlussendlich, Netzwerk auf dem MainThread zu erlauben, obgleich dieses Vorgehen nicht unbedingt der *best practice* entspricht. Perspektivisch sollten alle Netzwerkzugriffe über einen `AsyncTask` ausgeführt werden.

Quellenverzeichnis

[1] Etherpad Foundation: *„Etherpad"*. `http://etherpad.org/` (23.01.2014).

[2] Etherpad Foundation: *„Etherpad v1.3.0 Manual & Documentation"*. `http://etherpad.org/doc/v1.3.0/#index_http_api` (23.01.2014).

[3] Hollinger J: *java-etherpad-lite*. `https://github.com/jhollinger/java-etherpad-lite` (23.01.2014).