FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Speicherung von E-Mail Daten in einer dokumentbasierten Datenbank

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Information & Knowledge Management**

eingereicht von

**Karl Beranek, BSc.**
Matrikelnummer 0126665

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Prof. Dr. Andrew U. Frank

Wien, 23.06.2014

(Unterschrift Verfasser) (Unterschrift Betreuer)

# Erklärung zur Verfassung der Arbeit

Karl Beranek
12. Februarplatz 7/34/7
1190 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werke oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien am 23.06.2014

# Kurzfassung

Die Aufbewahrung von E-Mail Daten wird in vielen Fällen über Dateisystem-basierende Formate realisiert. Diese Formate erlauben eine hierarchische Gliederung der E-Mail Nachrichten, aber nur sehr eingeschränkte Möglichkeiten, die Nachrichten nach gewissen Gesichtspunkten zu organisieren. Einige Datenbank-basierende Methoden existieren, weisen jedoch den Nachteil auf, proprietär zu sein und keine offenen Schnittstellen anzubieten. Damit gilt für Datenbank- sowie auch Dateisystem-basierende Formate, dass diese nur schwer in Informationssysteme integrierbar sind.

Neue Möglichkeiten E-Mail-Daten effizient zu speichern werden durch dokumentbasierte Datenbanken eröffnet. Im Gegensatz zu relationen Datenbanken speichern dokumentbasierte Datenbankmanagementsysteme Daten in Form von Dokumenten, welche keinem festen Schema unterliegen müssen. In den letzten Jahren ist die Nutzung dieser dokumentbasierten Datenbanken erheblich gestiegen. Dennoch findet man, neben der Groupware Lotus Notes, keine weiteren Anwendungen die sich die Vorteile dieser Datenbanken bei der Aufbewahrung von E-Mail Daten zu nutze machen.

In meiner Diplomarbeit analysiere und vergleiche ich verschiedene Speicherungsarten von E-Mail Daten, um minimale Anforderungen an ein Mail-speicherndes System zu erheben. Weiters untersuche ich Eigenschaften und interne Strukturen des dokumentbasierten Datenbankmanagementsystems Apache CouchDB und deren Einsatz als Persistierungskomponente für E-Mail Daten.

Für die Realisierung und Überprüfung dieser Konzepte wurde ein Prototyp in der funktionalen Programmiersprache Haskell programmiert. Der Prototyp ist ein MIME-konformer Mail Client mit grafischer Benutzeroberfläche und speichert E-Mail Daten in der Apache CouchDB. Der Client zeigt gespeicherte E-Mail Nachrichten an und ermöglicht eine Suche nach bestimmten Merkmalen. Transformatoren für übliche Formate ermöglichen den Import aus bestehenden Mail-Archiven und erlauben es neue E-Mail Nachrichten über das Transportprotokoll POP3 von Mailservern zu holen.

# Abstract

E-Mail Data is mostly stored in filesystem-based data formats. These data formats allow arranging mails hierarchically, but serverly limit possibilites to organize mails to certain aspects. Some database-oriented methods are existing, however they have the disadvantage to be proprietary or don't provide open interfaces. It applies to both file-system and database-oriented formats, that these are difficult to integrate into information systems.

With document-oriented databases new possibilites arise to efficiently store e-mail data. Document-oriented database management systems store data in contrast to relational database system in documents. These documents are not subject to a fixed scheme. In the last few years the utilization of document-oriented databases have increased substantially. Still, beside the Groupware Lotus Notes no other applications are existing that harness the benefits of these databases in the field of persistence of e-mail data.

In my thesis I studied and compared different types of persistence of e-mail data and gathered minimal requirements for a system which stores mail. Consequently I examined features and internal structures of the document-based database management system Apache CouchDB and it's usage as a persistence component for e-mail data.

As proof of found concepts a prototyp was developed using the functional programming language Haskell. The developed prototyp is a MIME-compliant mail client with a graphical userinterface and stores e-mail data in the Apache CouchDB. The client displays stored e-mail messages and features a search function for different characteristics. Transformers for common formats enable the import of existing email archives, and the download of new e-mail messages from mail servers via the transport protocol POP3.

# Inhaltsverzeichnis

# Einleitung

In dokumentbasierten Datenbanken werden Daten in Form von Dokumenten gespeichert. Jedes Dokument steht für sich und unterliegt keinem festen Schema. Diese NoSQL Datenbanken weisen gravierende Vorteile in der Speicherung von großen, wenig strukturierten, vorwiegend textuellen Daten auf. Sie erlauben einfache Zugriffe von vielen Umgebungen durch ihre offenen Schnittstellen und bieten einen hohen Grad an Sicherheit der Daten durch Replikationsmechanismen und Revisionskontrolle.

In meiner Diplomarbeit stelle ich einen Einsatzzweck für dokumentbasierte Datenbanken vor: die Aufbewahrung von E-Mail Daten. Existierende proprietäre und offene Implementierungen für die Verarbeitung und Speicherung von Maildaten weisen unterschiedliche Vor- und Nachteile auf. Im Rahmen meiner Arbeit vergleiche ich verschiedene Speicherungsformen und deren Parameter. Ein weiterer Teil meiner Arbeit widmet sich der Entwicklung eines in der funktionalen Programmiersprache Haskell entworfenen Prototypen, der E-Maildaten in der CouchDB speichert. Konnektoren ermöglichen den Empfang von E-Mail Nachrichten von Mailserver über das Transportprotokoll POP3, den Import von MBOX-Mailarchiven sowie den SMTP-Versand von Mails. Eine grafische Benutzeroberfläche präsentiert die Funktionalitäten des Prototyps. Die Einbindung eines Frameworks für Volltext-Suche vereinfacht das Auffinden von E-Mailnachrichten in der Datenbank. Der Schwerpunkt meiner Arbeit ist die Feststellung der Machbarkeit eines solchen in Haskell programmierten Systems und die Untersuchung der Speicherung von Maildaten in CouchDB.

Meine Diplomarbeit ist wie folgt gegliedert: Im ersten Teil analysiere und vergleiche ich verschiedene Speicherungsformen von Mail-Daten. Teil II beschäftigt sich mit Grundlagen für die Realisierung des Prototypen welche im Teil III ins Detail erläutert wird. Kritisch diskutieren und vergleichen werde ich den Prototypen im letzten Kapitel. Zum Abschluss gebe ich eine Aussicht auf mögliche Funktionen einer Weiterentwicklung.

# Teil I.

# Formen der Speicherung von E-Mail Daten

Unter den existierenden Speicherungsarten für Maildaten wird zwischen datenbankbasierten Systemen und Systemen, die sich die Dateisystemstruktur zur Hilfe machen unterschieden. Bekannte Vertreter der erst genannten Form sind Microsoft Exchange und client-seitig, das offene, proprietäre Datenformat PST von Microsoft. Mail-Server unter Linux wie Postfix and Dovecot sowie Mail-Clients wie Mozilla Thunderbird oder Gnome Evolution nutzen Varianten von MBOX oder MailDir welche Daten direkt im Dateisystem ablegen.

Ein sehr großer Teil von E-Maildaten ist unstrukturiert, unterliegt keinem festen Schema, sondern besteht aus Klartext und binären Inhalten. Deswegen liegt es nahe die Vorteile der CouchDB bei der Aufbewahrung von großen, unstrukturierten Datenmengen zu nutzen und CouchDB als primären Speicher für E-Maildaten zu verwenden.

Wichtige Aspekte von E-Mail Daten, die für die Implementierung des Prototypen relevant sind, werden in den folgenden Kapiteln vorgestellt: das Format für Internet Mail, der "MIME Standard". Die darauf aufbauenden dateisystembasierenden Formate wie MBOX und MailDir werden kurz beschrieben. Im Anschluss werden wesentliche Ausprägungen von dokumentbasierten Datenbanken vor allem in Bezug auf die Speicherung von E-Mail-Daten beleuchtet und mit anderen Implementationen verglichen.

# 1. Multipurpose Internet Mail Extension

Multipurpose Internet Mail Extension ist der Standard zur Definition des Datenformat von Internet E-Mail, definiert in RFC 2045 bis RFC 2049. Es handelt sich dabei um eine Erweiterung des Standards für das Format von ARPA Internet Text Nachrichten, welcher 7bit E-Mail Nachrichten (Internetstandard RFC 822) beschreibt. Die Erweiterung ermöglicht den Austausch von nicht-ASCII Zeichen und binären 8-bit Inhalten wie z.B. Multimediadateien. Der MIME-Standard hat sich in vielen weiteren Bereichen bewiesen, so wird er ua. auch für über HTTP übertragene Inhalte eingesetzt. Eine Anforderung an die Spezifikation ist, dass MIME-Mails auch von nicht-MIME Mailservern und nicht-MIME Mail-Clients verarbeitet werden können, dh. dass keine Erweiterungen gegen die in den ursprünglichen Standard RFC 822 definierten Richtlinien verstoßen dürfen. [19]

Diese Erweiterungen haben dazu beigetragen das Format weltweit als primären Standard für das Datenformat von E-Mails zu etablieren. Im Folgenden werden die fünf Teile vorgestellt, in die die Spezifikation des MIME Standards aufgeteilt ist.

## 1.1. RFC2045: MIME Teil 1 “Format des Nachrichtenkörpers”

Grundsätzliche Email-Kopfzeilen wie “to”, “subject”, “from”, und “date” werden in der RFC 5322 definiert. MIME erweitert diese Gruppe von Kopfzeilen um weitere Attribute, damit neben der 7-Bit ASCII Zeichenkodierung auch weitere Kodierungen in Nachrichtenkörper und Kopfzeilen unterstützt werden und Multipart-Nachrichten ermöglicht werden. Multipart-Nachrichten sind Mail Nachrichten die aus mehreren Nachrichtenteilen, fortan als “Mailparts” bezeichnet, bestehen. Definiert werden diese zusätzlichen Kopfzeilen und die Syntax von MIME-Entitäten im ersten Teil der MIME Spezifikation. [19]

1. *MIME-Version:* Zeigt die Version der verwendeten MIME-Spezifikation und damit Konformität mit RFC2045 an;

2. *Content-Type:* Gibt den im Nachrichtenkörper der MIME-Entität enthaltenen Medientyp (spezifiziert in RFC2046) und optional den verwendeten Zeichensatz an, damit die empfangende Client-Applikation die Daten interpretieren kann, z.B.: "image/jpg". Ist die Kopfzeile Content-Type nicht angeführt, handelt es sich um ein Plaintext E-Mail (explizit definiert: "text/plain; charset=us-ascii");

3. *Content-Transfer-Encoding:* Gibt die Kodierung der Daten des Nachrichtenkörpers an; 8 Bit Zeichen oder binäre Daten können z.B. mittels SMTP nicht übertragen werden, da dies eine 7 Bit US-ASCII Kodierung voraussetzt - eine Rekodierung ist notwendig; ist die Kopfzeile nicht angeführt wird eine standardmäßige 7 Bit ASCII Kodierung angenommen;

4. *Content-ID:* Bei Verwendung von Multipart-Nachrichten erlaubt die Kopfzeile die Vergabe eines eindeutigen Identifiers für diesen Mailpart;

5. *Content-Description:* Kann deskriptive Informationen zu den im Nachrichtenkörper enhaltenen, binären Daten beinhalten;

## 1.2. RFC2046: MIME Teil 2 "Medien Typen"

Im zweiten Teil der Spezifikation wird das Typ-System von MIME beschrieben und in die Medien-Typen und -Untertypen eingeführt. Es wird unterschieden zwischen fünf diskreten Medientypen und den Kombinationstypen *Multipart* und *Message*. [21]

1. *Text:* Weist auf textuelle Information im Nachrichtenkörper der MIME-Entität hin. Der Untertyp "plain" gibt an, dass es sich um reinen Klartext handelt der keinerlei Formatierung enthält.

2. *Image, Audio, Video:* Der Nachrichtenkörper enthält Multimediadaten. Untertypen geben das verwendete Komprimierungsverfahren an.

3. *Application:* Der Nachrichtenkörper enthält binäre Daten oder Kontent, der in keine der anderen Kategorien passt. Ein eigenes Anwendungsprogramm wird benötigt um den Kontent zu öffnen oder zu verarbeiten.

4. *Multipart:* Bei Multipart-Nachrichten werden mehrere MIME-Entitäten in einem Nachrichtenkörper vereint. Die Entitäten bzw. Mailparts sind voneinander durch die im Parameter “Boundary” angegebene Zeichenkette getrennt. Jede MIME-Entität weißt eigene MIME-Kopfzeilen auf. Die Spezifikation erlaubt ausdrücklich mehrere in sich kaskadierte Multipart-Nachrichten wobei die Eindeutigkeit der “Boundary”-Zeichenketten gewahrt werden muss.

5. *Message:* Mit diesem Typ können Nachrichten innerhalb von Nachrichten gekapselt werden, wie im Fall von weitergeleiteten oder zurückgewiesenen Mail-Nachrichten.

## 1.3. RFC2047: MIME Teil 3 “Erweiterungen für Kopfzeilen”

Während die ersten Teile der Spezifikation die Verwendung weiterer Kodierungen neben der in RFC 822 spezifizierten US-ASCII Zeichenkodierung für den Nachrichtenkörper behandelt, beinhaltet der dritte Teil Erweiterungen für die Unterstützung von nicht-ASCII Zeichen in Kopfzeilen. Die Richtlinien führen eine Sequenz von ASCII Zeichen ein, dessen Auftreten in normalen Text höchst unwahrscheinlich ist und die ein sogenanntes Encoded-Word umschließen. Die Zeichenfolge “=?” leiten ein Encode-Word ein, während die Zeichenfolge “?=” es beenden. Ein Encoded-Word beinhaltet den verwendeten Zeichensatz, die Kodierung und die mit ASCII Zeichen kodierte Zeichenkette, jeweils voneinander getrennt mit den Zeichen “?” (siehe Listing 1).

```
"=?"<Zeichensatz>"?"<Kodierung>"?"<kodierte Zeichenfolge>"?="
```

**Listing 1:** Syntax eines Encoded-Word

Listing 2 zeigt ein Beispiel der Kodierung einer Kopfzeile, welche die Zeichenkette der Absenderidentifizierung (“From”) mit Zeichensatz Latin1 (“ISO-8859-1”) und der Kodierung Quoted-printable (“Q”) kodiert. Bei Quoted-printable werden 8-Bit Zeichen mit einem “=” gefolgt vom hexadezimalen Wert des Zeichens repräsentiert. [20]

```
From: =?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?=
```

**Listing 2:** Beispiel Encoded-Word

## 1.4. RFC2048: MIME Teil 4 “Registrierungsabläufe”

Der 4. Teil der Spezifikation beschreibt administrative und organisatorische Abläufe für die Registrierung neuer Medientypen oder Kodierungsverfahren bei der IANA, der Internet Assigned Numbers Authority. [18]

## 1.5. RFC2049: MIME Teil 5, “Medien Typen”

Einige Regeln und Richtlinien, die in den MIME Spezifikationen beschrieben werden, sind nicht genau definiert und erlauben einen großen Implementationsspielraum. Um eine grundsätzliche Interoperabilität zu gewährleisten wird in der RFC2049 die MIME-Konformität beschrieben, die die Mindestanforderungen an MIME E-Mail Clients festlegt. [17]

1. Jede erstellte Mail-Nachricht muss die Kopfzeile MIME-Version mit den Wert 1.0 enthalten.

2. Der Nachrichtenkörper muss entsprechend dem in der Kopfzeile Content-Transfer-Encoding angegebenen Verfahren dekodiert werden. Eine 7-Bit, 8 Bit und binäre Transformationen muss erkannt werden.

3. Unbekannte Kodierungen werden als Typ ”application/octet-stream“ behandelt.

4. Nachrichtenkörper, die nicht im Klartext übertragen wurden, dürfen nicht als Rohdaten angezeigt werden. Zumindest Klartext-Nachrichten sollen angezeigt und versendet werden können.

5. Unbekannte Parameter der Kopfzeile Content-Type sollen ignoriert werden.

6. Für unbekannte Medientypen muss die Option bestehen die Daten in eine Datei zu schreiben oder an eine alternative Applikation übergeben zu werden.

7. Konforme Mail-Clients die auch das Senden und Empfangen von E-Mails unterstützen, die nicht der MIME-Spezfikation unterliegen, dürfen nur US-ASCII zeichenkodierte E-Mails versenden.

8. Die Unterstützung von Encoded-Words in den registrierten Enkodierungen Quoted-Printable und Base64 muss gegeben sein.

# 2. MBOX Datenbank

Der MBOX Standard beschreibt die Speicherung von mehreren E-Mails in einem Postfach. Formal beschrieben definiert der Standard das Format zur Speicherung von linearen Sequenzen von einer oder mehreren MIME E-Mail Nachrichten in einer Datei. Historisch hat das Format seinen Ursprung in UNIX Betriebssystemen, wird allerdings wegen der einfachen Handhabung auch auf weiteren Plattformen verwendet. Durch das Fehlen einer verbindlichen Spezifikation hat sich eine Familie von untereinander inkompatiblen

MBOX Formaten entwickelt. Das MBOX Dateiformat baut auf die MIME Spezifikation auf. Im Folgenden wird der grundsätzliche Aufbau beschrieben.

Der Anfang einer Nachricht im MBOX Datenformat wird mit der Trennzeile eingeleitet die den Absender identifiziert. Die Zeile besteht aus der Zeichenkette “From”, einem einzelnen Leerzeichen, der Absender-Emailadresse und dem Zeichen für Zeilenende. In manchen Implementationen ist auch ein Zeitstempel, der den Zeitpunkt des Empfangs angibt, vorgesehen. In den folgenden Zeilen sind weitere Kopfzeilen-Informationen wie Zieladresse, Sendedatum oder Kodierung enthalten. Eine Leerzeile markiert das Ende der Kopfzeilen und den Start des Nachrichteninhalts in 7 Bit Kodierung. Ist die Zeichensequenz “From” am Anfang eine Zeile im E-Mailkontent enthalten, wird diese durch “¿From” ersetzt um die Eindeutigkeit der “From”-Kopfzeile zu wahren. Eine Leerzeile und das EndOfFile Zeichen markiert das Ende des E-Mailkontents und das Dateiende.

Ist eine weitere Mail-Nachricht in der MBOX-Datei enthalten, folgt der Leerzeile die Trennzeile mit der “From”-Zeichenkette und weitere Kopfzeileninformationen (siehe Listing 3). [16] [14]

```
From: joe@example.com
To: mike@example.com

Hi Mike
looking forward to hear from you!
Best regards, Joe

From: leo@example.com
...
```

**Listing 3:** MBOX Mail

Ein Nachteil bei Verwendung des MBOX Datenformats entsteht dadurch, dass während Änderungen oder Neuerstellungen einer Mail die MBOX-Datei gesperrt ist. Gleichzeitig,

schreibender Zugriff von mehreren Benutzern oder Prozessen ist somit nicht möglich.

# 3. MailDir Datenbank

Dieses Format verwendet im Gegensatz zu MBOX für jede einzelne Mail-Nachricht eine eigene Datei mit eindeutigen Dateinamen. Mehrere Mail-Dateien in einem Verzeichnis bilden eine Mailbox. Die Spezifikation von MailDir sieht fix definierte Verzeichnisse für gerade eingehende E-Mails, ungelesene und gelesene E-Mails vor. Desweiteren beschreibt sie Verfahren, um eindeutige Dateinamen für die Nachrichten-Dateien zu generieren, sowie die Möglichkeit, Nachrichten mit Markierungen wie “beantwortet”, “gelöscht” und dergleichen zu markieren. [6]

Es gelingt somit das Sperrverhalten von MailDir Datenbanken minimal zu halten. Ein Problem dieses Formats ist, dass bei großen Maildatenbeständen, durch die 1:1 Zuordnung von Nachrichten zu Dateien, sehr viele Dateien im Dateisystem gespeichert und verwaltet werden müssen. Aufgrund des einfach Konzepts konnte sich MailDir jedoch als Format für ein breites Spektrum von E-Mail speichernden Systemen etablieren. [2]

# 4. Personal Storage Table

Als ein der Vertreter der datenbank-basierten Speicherungsarten für Mail-Daten zählt das offene, proprietäre Datenbankformat Personal Storage Table (PST). Dieses wird von Microsoft E-Mail Clients verwendet und kann neben E-Mail Nachrichten auch weitere Informationen wie Kalender-, Kontakt- und Notizdaten speichern. Die Datenstruktur der Datenbank wird mit B-Bäumen realisiert und ermöglicht, durch Verwendung von 64 Bit Zeigern, Größen über 50GB.

Probleme ergeben sich bei bei dieser Speicherungsart bei der gemeinsamen Verwendung

im Netzwerk. Ein bearbeitender Client sperrt die gesamte PST-Datei. Replikationen oder Sicherungen sind nur in definierten Backup-Fenster möglich. Die Datenbank-Datei ist von der Clientanwendung ständig geöffnet, dadurch kann es bei Netzwerkfehlern zu Korruption der Datei kommen. Nur mittels zeitaufwendiger Reparatur und Konsistenzprüfung kann das Problem beseitigt werden können. [13]

# 5. Vergleich der vorgestellten Speicherungsarten

Wie in den vorangegangen Kapiteln beschrieben, speichert das unmodifizierte MBOX-Format, E-Mails in einem Postfach, kodiert im ursprünglichen MIME Format in einer zusammenhängenden Zeichenkette, in einer Datei ab. Dies ermöglicht ein schnelles Hinzufügen von neuen E-Mails und eine schnelle Suche nach E-Mails innerhalb eines Postfachs. Ein Nachteil von MBOX ist, dass Änderungen einer E-Mail in einer MBOX-Datei, die gesamte Datei sperrt und weitere Zugriffe unterbindet. Dieses Problem verhindert den Einsatz von MBOX-Dateien in einer Mehrbenutzerumgebung.

Das Mailformat MailDir arbeitet mit einer 1:1 Zuordnung von E-Mails und Dateien. Das betroffene E-Mail muss bei Zugriffen gesperrt werden, andere E-Mails im gleichen Postfach sind für weitere Zugriffe verfügbar. Dieses minimale Sperrverhalten von Dateien macht das Format für die Verwendung in einer Mehrbenutzerumgebung tauglich. Die riesige Anzahl an Dateien erschwert allerdings die Replikation und die Administration und kann von manchen Netzwerk-Dateisystemen nicht effizient verarbeitet werden. Zusätzlich wird die Indexierung und Suche verlangsamt. [6]

Auch das PST Datenbankformat sperrt die Datendatei bei Verwendung und erlaubt keinen gleichzeitigen Mehrbenutzer-Zugriff.

Alle beschriebenen Formate erlauben eine hierarchische Gliederung der E-Mail Nachrichten, aber nur sehr eingeschränkte Möglichkeiten, die Nachrichten nach gewissen Gesichtspunkten zu organisieren. Auch der gleichzeitige Zugriff von anderen Appli-

kationen oder Benutzern ist eingeschränkt. Durch die Einführung einer zusätzlichen Persistierungsschicht mit einer dokument-basierenden Datenbank können diese Einschränkungen verhindert werden. Der Einsatz der dokument-basierenden Datenbank CouchDB bringt desweiteren folgende Vorteile: [8]

**Datenorganisation und Zugriff:** Die Datenbank sorgt per se für eine optimale Datenhaltung und -Organisation. Der gleichzeitige Zugriff auf E-Mail Nachrichten in einer Mehrbenutzerumgebung ist durch die Funktionsweise der CouchDB gegeben - keine Sperrmechanismen!

**Datensicherheit:** Irrtümlich gelöschte E-Mail Nachrichten können wieder hergestellt werden, da alle Daten in der CouchDB der Versionierung unterliegen. Konsistente Backups können jederzeit erfolgen.

**Verfügbarkeit und Skalierbarkeit:** Die Replikation ermöglicht die Verwendung über mehrere CouchDB Instanzen. Größen-Limits von CouchDB ergeben sich nur auf Grund von Grenzen der verwendeten Speichernetzwerke.

**Schnittstellen:** Zugriffe erfolgen über eine REST/HTTP Schnittstelle.

**Indexierung:** Durch die Integration von zusätzlichen Frameworks, wird eine effiziente Indexierung und Volltextsuche von Daten in CouchDB gewährleistet (siehe Apache Lucene, Kapitel 7).

Eine

# Teil II.

# Grundlagen

Folgende Kapitel führen in die Grundlagen von Frameworks und Konzepte ein auf die im dritten Teil ”Realisierung des Prototyps“ Bezug genommen werden. Kapitel 6 stellt die dokumentbasierte Datenbank CouchDB vor welche als Speicherungsform von E-Maildaten eingesetzt werden soll. Für die Erstellung von Suchindezes in CouchDB Datenbanken wird Apache Lucene eingesetzt werden, welches im Kapitel 7 erläutert wird. Die Verwendung des MIME Standards in einem Prototyp programmiert in Haskell birgt Herausforderungen. Diese sowie das Toolkit GTK+ welches zur Gestaltung von grafischen Benutzeroberfläche eingesetzt wird werden in den letzten Kapiteln des Grundlagen-Teils angeführt.

## 6. CouchDB

CouchDB ist eine Dokument Datenbank, entwickelt in der Programmiersprache Erlang, die eine RESTful API bietet, dh. die Kommunikation läuft über über das zustandlose HTTP-Protokol und basiert auf den Methoden POST, GET, PUT und DELETE. [9]

Die CouchDB zeichnet sich durch eine einfache Anwendung und leicht nachvollziehbare Konzepte aus. Bewährte Ideen von Web Architekturen und von verteilten Systemen wurden bei der Entwicklung übernommen. CouchDB bietet eine hohe Fehlertoleranz bei der Kommunikation über mehrere Datenbank-Knoten. Schon beim Design von CouchDB wurden Probleme bei Netzwerkverbindungen, Fehler bei Schreibvorgängen auf Festplatten oder verzögerte Zugriffe auf andere Systeme berücksichtigt. Design-Entscheidungen, detailliert in den Kapiteln 6.3 und 6.4 wurden getroffen, unvorhersagbare Ereignissen zu begegnen und ein hoch tolerantes System zu schaffen. Die hohe Verfügbarkeit und

Fehlertoleranz von Standard Hardwarekomponenten ist der Namensgeber von CouchDB, welche ausgeschrieben “Cluster of unreliable commodity hardware data base” heißt.

In den folgenden Kapiteln befasse ich mich mit den allgemeinen Eigenschaften und Funktionen der dokumentbasierten Datenbank Apache CouchDB. Hauptmerkmale von Dokumenten, interne Datenstrukturen, Konsistenz und Möglichkeiten der Replizierung werden angeführt. [8]

## 6.1. CouchDB Dokumente

Die zentralen Datenstrukturen in CouchDB sind CouchDB-Dokumente. In diesen Datenstrukturen werden in sich geschlossene Informationen gespeichert. Alle relevanten Informationen befinden sich in einer Einheit, wie dies zum Beispiel bei einem realen Dokument, einer Rechnung der Fall ist. In der Datenbank Terminologie spricht man von “self-contained data”, in sich geschlossene und unabhängige Datenstrukturen.

Bei relationalen Datenbanksystemen ist das oberste Ziel, Daten zu normalisieren, um Redundanzen zu vermeiden. In CouchDB werden Daten primär in denormalisierter Struktur gespeichert; ein abgerufenes Dokument für sich alleine beinhaltet aussagekräftige Informationen. [8]

### JavaScript Object Notation

CouchDB Dokumenten werden in der CouchDB mit dem Datenaustauschformat JSON “JavaScript Object Notation” dargestellt. Die JSON Syntax definiert ein simples Textformat für die Speicherung und Übertragung von strukturierten Daten über Netzwerkverbindungen. Das Datenaustauschformat JSON ist im Gegensatz zu XML typisiert und unterscheidet zwischen folgenden primitiven Datentypen: Zeichenketten, Zahlen, boolsche Werte und dem Typ Null sowie den strukturierten Datentypen: Objekte und

Arrays. Ein Objekt stellt eine ungeordnete Sammlung von Schlüssel/Werte-Paaren dar, ein Array eine geordnete Liste von Werten. [15] Inhalte eines CouchDB Dokuments werden als Objekte repräsentiert. In Listing 4 ist ein in JSON dargestelltes einfaches Dokument exemplarisch angeführt: [8]

```
{
    "_id": "01f1f39ca3216bfef7f077c6c20005f7",
    "_rev": "5-a47ef17f2ea26a5df1dede9865142cc5",
    "type": "mail",
    "from": "karl.beranek@gmx.net",
    "body": "Test"
}
```

**Listing 4:** CouchDB Dokument in JSON Syntax

### Universally Unified Identifier

Jedes Dokument in CouchDB enthält ein Attribut "ID" in der ein Universally Unified Identifier (UUID) gespeichert wird. UUIDs sind zufällig generierte Zeichenketten, dessen Einzigartikeit zwar nicht garantiert ist, die Wahrscheinlichkeit, dass eine gleiche Zeichenkette erstellt wird ist aber verschwindend gering. Somit stellen UUIDs eine einfache Möglichkeit dar, Dokumente auch systemübergreifend eindeutig zu identifizieren. Ein weiteres, in jedem CouchDB-Dokument enthaltenes Attribut, ist die Revisions-ID, die unterschiedliche Versionen des gleichen Dokuments kennzeichnen. [8]

### Revisions-ID

Alle Dokumente in CouchDB sind versioniert. Änderungen in einem Dokument rufen jeweils eine Neuerstellung des Dokuments unter einer neuen Revision hervor. Die Revisions-ID (siehe Listing 4 Feldname "_rev" ) besteht aus einem Integerwert, gefolgt von einem Bindestrich und einem MD5-Hash-Wert, der aus Eigenschaften des

Dokuments berechnet wird. Der Integerwert entspricht der Anzahl der Änderungen des Dokuments. [8]

## 6.2. Design Dokumente

In CouchDB existiert eine spezielle Form von Dokumenten, sogenannte Design Dokumente. Diese Dokumente speichern Funktionscode und sind im Gegensatz zu normalen Dokumenten strukturiert. Mit ihnen lassen sich Funktionalitäten wie Abfragen, Validierungsfunktionen oder Transformationen realisieren aber auch das Erstellen einfacher Web-Applikationen wird ermöglicht.

Die Dokument-IDs von Design Dokumenten beginnen immer mit der Zeichenkette "_design/". Sie weisen dieselben Eigenschaften wie andere CouchDB Dokumente auf und unterliegen einer Versionierung. In Listing 5 ist ein Beispiel eines Design Dokuments mit einem CouchDB-View und einer Validierungsfunktion angeführt. [8]

```
{
   "_id": "_design/example",
   "_rev": "12-8fd98930888f6434ee79504665682ba8",
   "views": {
       "foo": {
            "map": "function(doc)
                {if (doc.type == "mail")
                    { emit (doc.from, doc.body)}
            }"} },
   "validate_doc_update": "function(newDoc, savedDoc, userCtx)
      {
        if (newDoc.type = "mail") {
            // validation logic for mails
       } }"
}
```

**Listing 5:** DesignDocument

### Abfragen in CouchDB: Views

Im Gegensatz zu relationalen Datenbanksystemen werden in CouchDB ausschließlich vordefinierte Abfragen verwendet, sogenannte Views. Mit Hilfe von Views können Dokumente gefiltert, Daten extrahiert und Berechnungen mit in den Dokumenten enthaltenen Daten durchgeführt werden.

Für den Aufbau eines solchen Views werden Map und Reduce Funktionen in Design Dokumenten definiert (siehe Code-Listing 5).

**Map-Funktionen** Map-Funktionen sind seiteneffekt-freie, in JavaScript geschriebene Funktionen, die einmal mit jedem der bestehenden Dokumente als Argumente aufgerufen werden. In der Funktion ist festgelegt, welche Dokumente selektiert und welche Eigenschaften als Schlüssel/Werte-Paar ausgegeben werden. Da Map-Funktionen nur von Informationen innerhalb des Dokuments abhängig sind, werden diese maximal einmal pro Dokument ausgeführt. Die resultierende Menge an Schlüssel/Werte-Paaren, auch "Intermediate Result" genannt, werden wie auch Dokumente in CouchDB, als B-Tree gespeichert.

**Reduce-Funktionen** Reduce Funktionen aggregieren und "reduzieren" eine Menge von Dokumenten. Sie verarbeiten das Intermediate Result der Map Funktion, indem Werte aus den Schlüssel/Werte-Paaren berechnet werden. Dies führt zur Bildung des "View Results". Wie bereits beschrieben, wird bei der Map-Funktion jedes Dokument in Isolation behandelt. Reduce Funktionen erlauben Dokument-übergreifende Berechnungen wie zum Beispiel: Summenbildungen aus Werten verschiedener Dokumente oder die Ausgabe von eindeutigen Werten (vergleiche DISTINCT bei SQL). [8]

### Validierungsfunktionen

Validierungsfunktionen stellen sicher, dass auschließlich Dokumente, die definierte Bedingungen erfüllen, in der Datenbank gespeichert werden. JavaScript Funktionen werden in Design Dokumenten hinterlegt, die das Vorhandensein oder Inhalte von bestimmten Attributen, in neuen Dokumenten entsprechend ihres Datentyps, überprüfen. In CouchDB existieren verschiedene Methoden um Dokumente einen Datentyp zu zuweisen um diese korrekt zu validieren: Eine Variante bedingt die Einführung eines neuen Attributs mit dem Namen z.B: “Type” dessen Wert dem Datentyp des Dokuments entspricht. Eine weitere Variante bedient sich des Konzepts des *Duck Typing* und prüft vorhandene Attribute eines Dokuments, um dessen Typ zu identifizieren und das Dokument zu validieren. Listing 5 zeigt ein einfaches Beispiel einer Validierungsfunktion im Attribut *validate_doc_update* nach erst genannter Variante. [8]

### CouchDB als Applikationsserver

CouchDB auch als Applikationsserver agieren. Mittels der Show Funktion in Design Dokument wird das Rendering von Dokumenten bestimmt. Dokumente können im Browser als benutzerdefiniertes Format oder zum Beispiel als HTML-Seite ausgegeben werden. Die List-Funktion ermöglicht Transformationen und Aggregationen auf mehrere Dokumente. Komplette Standalone Applikationen, sogenannte CouchApps, können in Design Dokumenten gespeichert und betrieben werden. Die Applikationen bleiben als CouchDB Dokumente bestehen und behalten alle Vorteile derselben (z.B. Replikation, Revisionskontrolle). [8]

## 6.3. Interne Datenstruktur: B+ -Bäume

In der CouchDB werden alle Dokumente sowie die Schlüssel/Werte-Paare eines Views in B+ -Bäumen gespeichert. In B+ -Bäumen werden die Datensätze in den Blättern

abgelegt, innere Knoten beinhalten lediglich Indizes. Möchte man den Datensatz zu einem Schlüssel finden, bewegt man sich durch die inneren Knoten bis zum Knotenblatt. Ein typischer B-Baum erreicht auch bei Millionen von Einträgen nur einstellige Höhen. Dies führt bei der Speicherung auf Festplatten zu wenig Suchvorgängen des Lesekopfs und gesamt zu kurzen Zugriffszeiten.

Wie bereits angeführt wird der B-Baum zu einem View-Resultat nur einmal generiert und jede nachfolgende Abfrage liest den bereits erstellten B-Baum. Werden nun Eigenschaften eines Dokuments geändert, markiert CouchDB jene Reihen im View-Resultat als ungültig die dieses Dokument betreffen, berechnet die Reihen neu und fügt sie wieder den B-Baum hinzu. Die Erstellung neuer Dokumente initiert ebenso die Neugenerierung einzelner Reihen im B-Baum des View-Results. [28]

Die Implementation von B-Bäumen in CouchDB enthält folgende Erweiterungen:

1. *Multiversion Concurrency Control:* Eine zusätzliche Komponente für Versionskontrolle ist implementiert worden damit Lese- und Schreibzugriffe ohne Sperrmechanismen funktionieren. Alle Schreibvorgänge innerhalb einer Datenbank werden hintereinander abgearbeitet und sperren keine Lesevorgänge. Das vielfache Lesen und Schreiben von Daten aus der Datenbank kann aufgrund des Append-only Designs gleichzeitig und konsistent erfolgen.

2. *Append-only Design:* Neue und geänderte Daten werden immer nur am Ende der Datenbankdatei hinzugefügt und der Wurzelknoten aktualisiert. Da alte Bereiche der Datenbank nie geändert werden, kann ein Zeiger nie auf ungültige Bereiche zeigen. Selbst ein veralteter Zeiger zeigt immer auf konsistente Momentaufnahmen der Datenbank.

Letztere Eigenschaft verschafft dem CouchDB Design auch die Kennzeichnung “crash only”, da beim Beenden der Datenbankdienste keine Routinen aufgerufen werden, die das Stoppen des Dienstes einleiten, wie es bei anderen Datenbanksystemen der Fall ist.

Der CouchDB Prozess wird beim Stoppen lediglich terminiert. [8]

## 6.4. Eventual consistency

CouchDB bietet eine hohe Verfügbarkeit und Ausfallstoleranz auf Kosten der vollständigen Konsistenz und akzeptiert eine sogenannte “Eventual Consistency”. Die “Eventual consistency” besagt, dass Daten nach einer hinreichend langen Zeit konsistent sind. Unterschiedliche Strategien um diesen Anforderungen nachzukommen sind im CAP Theorem festgehalten, welches im Folgenden beschrieben ist. [8]

### Cap Theorem

Das Cap Theorem beschreibt Strategien für verteilte Datenbanksysteme und unterscheidet dabei zwischen folgenden drei Anforderungen:

**Konsistenz:** Alle Datenbank Clients greifen auf idente Datensätze zu - auch bei gleichzeitigen Änderungen.

**Verfügbarkeit:** Alle Datenbank Clients können zu jeder Zeit auf eine Version der Datensätze zugreifen.

**Ausfallstoleranz:** Die Datenbank kann über mehrere Server verteilt werden. Das System bleibt verfügbar auch beim Ausfall eines oder mehrerer Server.

Drei dieser Belange sind nie gleichzeitig und vollständig erreichbar. Ist beispielsweise vollständige Konsistenz der Daten erforderlich, müssen Änderungen von Daten erst auf alle Server synchronisiert werden, bevor Daten wieder gelesen oder geschrieben werden. Dies bedeutet allerdings, dass die Verfügbarkeit zu Gunsten der Konsistenz eingeschränkt wird [8].

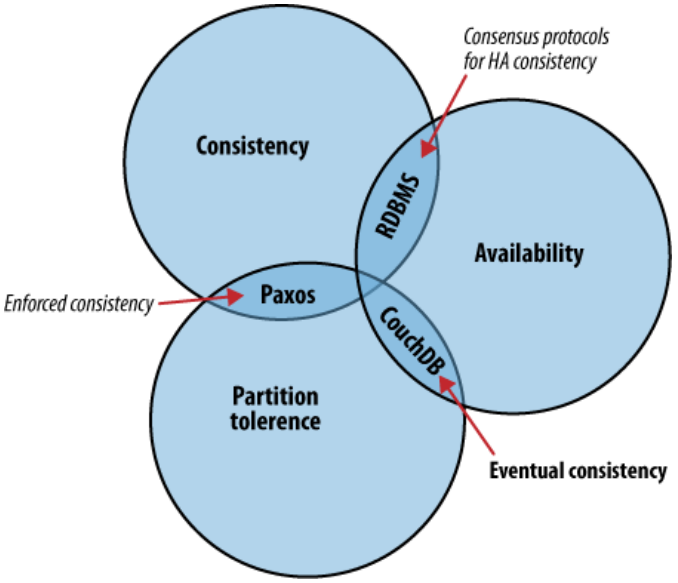


**Abbildung 1:** Cap Theorem

Abbildung 1 illustriert das Cap Theorem. Entwickler von verteilten Anwendungen müssen wählen welche Anforderungen priorisiert und implementiert werden. Die Abbildung führt als Beispiel für eine Datenbank in der Hochverfügbarkeit und Konsistenz an erster Stelle stehen, relationale Datenbank-Managementsysteme (RDBMS) an. CouchDB befindet sich in der Abbildung zwischen Verfügbarkeit und Ausfallstoleranz und bietet eine “Eventual consistency”.

### Eventual consistency

Die Entwickler von CouchDB verfolgen den Ansatz, dass jede Komponente eines verteilten Systems autark ist und Entscheidungen ausschließlich aufgrund lokaler Zustände getroffen werden können. Dies hat zur Folge, dass das System bei Schreibzugriffen nicht auf Abstimmung mit anderen Netzknoten wartet, sondern die Änderungen sofort übernommen werden. Der Schreibzugriff erfolgt damit immer mit maximal möglicher Geschwindigkeit. In Konflikt stehende, gleichzeitige Änderungen von Daten unterschiedlicher CouchDB Instanzen werden anschließend vom Datenbanksystem gelöst oder an die Applikation weitergegeben. Demzufolge erhält CouchDB einen großen Performancegewinn und eine hohe Verfügbarkeit auf Kosten der sofortigen Konsistenz und erreicht

nach einer systemweiten Konfliktbeseitigung eine “Eventual Consistency”. [8]

## 6.5. CouchDB Attachments

CouchDB Attachments, im Folgenden CouchDB Anhänge genannt, ermöglichen es binäre Daten speichereffizient in CouchDB zu speichern. Jedem Dokument können ein oder mehrere Anhänge beigefügt werden. Zu jedem Anhang werden die Metadaten Name, Content-Type und die Anzahl der Bytes gespeichert. Wird der Anhang per HTTP Request aufgerufen, erstellt CouchDB den entsprechenden HTTP Header, damit die Datei zum Beispiel im Browser korrekt dargestellt werden kann. [8]

Es wird zwischen Standalone Attachments, Inline Attachments und Multiple Attachments unterschieden. Standalone Attachments werden als eigenständige Dokumente ohne weitere Attribute gespeichert. Inline Attachments werden zu bestehenden Dokumenten angehängt und erhalten den reservierten Attributnamen _attachments. Sie werden mit Namen, Kontenttyp, Größe und den Base64 kodierten Daten als JSON Struktur gespeichert (siehe Listing 6). Multiple Attachments können entweder in JSON Struktur eingebetteten JSON Objekten dargestellt werden oder im *MIME multipart/related* Format. [27]

```
{ "_id":"attachment_doc",
  "_attachments":
  {
    "example.txt":
    {
      "content_type":"text\/plain",
      "length":40,
      "data": "VGhpcyBpcyBhIGJhc2U2NCBlbmNvZGVkIHRleHQ="
} } }
```

**Listing 6:** CouchDB Anhang

## 6.6. Replikation

Die gute Skalierbarkeit von CouchDB wird durch unterschiedliche Features erreicht. Ein wesentlicher Faktor sind die Mechanismen zur inkrementellen Replikation von Datenbanken auf weitere CouchDB-Server. Dabei können alle Dokumente von einer CouchDB Instanz, auf eine weitere repliziert werden. Zu übertragende Dokumente werden anhand ihrer Revisions-ID identifiziert. Existiert das Dokument mit gleicher Revision auf beiden Seiten wird es nicht übertragen. Nur geänderte, neue und gelöschte Dokumente werden repliziert. Die Replikation kann unidirektional, bidirektional oder kontinuierlich stattfinden. Mit Hilfe von Design-Dokumenten kann ein Filter der zu replizierenden Dokumente spezifiziert werden.

CouchDB unterscheidet nicht ob sich die Ziel Datenbank lokal oder auf einem entfernen Standort befindet. Die Erstellung eines neuen Replikations-Auftrags sowie die Übertragung zwischen Quell- und Zielserver basiert, wie jegliche Kommunikation bei CouchDB, auf Basis von HTTP Befehlen. Eine stetige Synchronisierung von zwei Datenbanken kann durch die Einrichtung einer bidirektionalen Replikation, mit Hilfe von zwei Replikationsjobs, mit getauschten Ziel- und Quell-Server erreicht werden.

Mit Hilfe der Replikations-Mechanismen ist es möglich ein CouchDB Cluster mit mehreren Systemen aufzubauen, um den Anforderungen für Hochverfügbarkeit und Lastenverteilung bei erhöhter Last nachzukommen. [8]

# 7. Apache Lucene

Apache CouchDB Map und Reduce Funktionen ermöglichen eine gezielte Filterung und Ausgabe von CouchDB-Dokumenten. Eine Volltext-Suche, die definierte Dokument-Attribute und Anhänge durchsucht, kann nur aufwendig mit diesen Funktionen ermöglicht werden. Das Apache Lucene Framework, im folgenden kurz Lucene genannt, stellt

Funktionalitäten zur Verfügung, die eine Volltextsuche nach Attributen und Anhängen ermöglicht. Lucene ist ein Open-Source Projekt, das eine plattformübergreifende, in Java geschriebene Suchmaschinen-API entwickelt hat. Lucene bietet Features wie automatische Indexierung, hoch performante Suchabfragen und priorisierte Suchresultate.

## 7.1. Indexierung

Die wichtigste Funktionalität von Lucene ist die Erstellung eines Index. Lucene stellt für unterschiedliche Datenformate Parser bereit um Text extrahieren zu können. [30] Bevor ein Index erstellt wird, muss dieser Text aufbearbeitet werden. Dabei kommen unterschiedliche Methoden zum Einsatz. Der Inputtext ist in eine Anzahl von Tokens umzuwandeln. Stop-Words, Worte die keinen aussagekräftigen Inhalt haben, werden entfernt; um morphologische Varianten von Worten auf ihren Wortstamm zu reduzieren werden sogenannte “Stemming”-Methoden eingesetzt; Großbuchstaben und Sonderzeichen, wie Umlaute oder akzentuierte Zeichen, werden in Kleinbuchstaben und einfache ASCII Zeichen umgewandelt.

In Listing 7 sind Beispiele für deutsche Stop-Words, die aus den Inputtexten entfernt werden, angeführt: [12]

```
aber, als, am, auch, bei, darum, dass, deine, deshalb,
durch, fuer, hatte, hier, im, ist, ja, jede, jetzt, muss,
nach, oder, sind, sollen, und, unter, vom, vor, wann,
was, weiter, wenn, werden, wie, wir, wird, wo, zu, ueber
```

**Listing 7:** Deutsche Stop-Words

“Stemming”-Algorithmen ermitteln die kleinsten sinngebenden Einheiten, sogenannte Tokens. Bei diesem Verfahren geht ein Teil von Information verloren, da sich manche Worte mit unterschiedlichen Bedeutungen auf den selben Stamm reduzieren lassen. Die Vorteile, die meisten Worte mit gleichen Konzept als solche zu identifizieren, haben

| Wort | Stamm | Wort | Stamm |
|---|---|---|---|
| aufeinander | aufeinand | kategorie | kategori |
| aufeinanderfolgen | aufeinanderfolgen | kategorien | kategori |
| aufeinanderfolgende | aufeinanderfolg | kategorisch | kategor |
| aufeinanderfolgender | aufeinanderfolg | kategorischer | kategor |
| aufeinanderfolgten | aufeinanderfolgt | katholik | kathol |
| auferwecken | auferweck | katholische | kathol |
| auferweckt | auferweckt | katholischen | kathol |
| auffallen | auffall | kauf | kauf |
| auffallender | auffall | kaufe | kauf |
| auffälligen | auffall | kaufen | kauf |
| auffälliges | auffall | käufer | kauf |

**Tabelle 1:** Beispiele Stemmer Algorithmus

sich jedoch bewiesen. Tabelle 1 führt einige Beispiele von Worten und dem gekürzten Wortstamm mit Hilfe des deutschen Porter-Stemmer-Algorithmus an. [12]

Indezierte Daten können in Lucene in benannten Attributen gespeichert werden. Dadurch wird die Suche nach auschließlich in diesen Attributen enhaltenen Text ermöglicht.

Lucene unterscheidet zwischen verschiedenen Feldtypen im Lucene Index: [7]

**Keyword:** Der Inputtext wird indiziert und unverändert im Index gespeichert. Inhalte können gesucht und im Suchresultat ausgegeben werden. Diese Art von Attributen eignet sich für Inhalte, dessen Inhalt auch bei geringfügigen Veränderungen verloren geht, zum Beispiel URLs oder Identifikationsnummern.

**UnStored:** Der Text wird in Tokens umgewandelt und indiziert, aber nicht im Index gespeichert; geeignet für große Dokumente deren ursprüngliche Form nicht direkt bei Suchresultaten benötigt werden.

**UnIndexed:** Der Text wird weder in Tokens umgewandelt noch indiziert und wird unverändert im Index gespeichert; es kann nicht danach gesucht werden, kann aber bei Suchresultaten mit angezeigt werden, z.B. Dateisystempfade.

**Text:** Der Inputtext wird in Tokens umgewandelt, indiziert und komplett im Index gespeichert, z.B. Titel oder Zusammenfassungen.

## 7.2. CouchDB-Lucene

Für eine effiziente Volltext Suche in CouchDB, kann auf die vom CouchDB-Lucene Projekt zur Verfügung gestellte Bibliothek zurückgegriffen werden. Nach der Installation läuft CouchDB-Lucene in einer Java Virtual Machine als Daemon und kann über Suchanfragen per HTTP angesprochen werden. Änderungen von Dokumenten in CouchDB werden vom Indizierungs-Prozess erkannt und im Index angepaßt. Die Indexfunktion wird in JSON-Syntax in einem CouchDB-Design-Dokument festgelegt. [4]

```
"fulltext": {
    "by_subject": {
        "defaults": { "store":"yes" },
         "index":"function(doc) { var ret=new Document();
             ret.add(doc.subject); return ret
}   }    }    }"
```

**Listing 8:** Indexfunktion im DesignDokument

Das in Listing 8 angeführte Design-Dokument definiert einen Index mit dem Namen “by_subject”, den Standard-Parameter “store” mit Wert “yes” und die zugehörige Indexfunktion. Diese Funktion erstellt ein neues CouchDB-Dokument, fügt aus den übergebenen Dokumenten den Inhalt des Attributs “Subject” hinzu und returniert das erstellte Dokument.

Zu jedem Attribut können Parameter angegeben werden. Sie legen fest unter welchen Namen der Inhalt des Attributs im Index gespeichert wird (“name”), von welchen Typ der Inhalt ist (“type”), ob die Daten im Index mitgespeichert werden (“store”) und ob und mit welchen Verfahren der Inhalt indexiert werden sollen (“index”). Pro Attribut kann ein Faktor angegeben werden, mit dem der Score des Attributs erhöht wird. Der Score ist ein normalisierter Wert, der die Ähnlichkeit zwischen Suchanfrage und Treffer angibt.

Suchanfragen efolgen über URLs und beinhalten den Datenbanknamen, das Design-Dokument, den Indexnamen, Suchparameter und den gesuchten Begriff: `http://localhost:5984/dbname_fti/search/by_subject?q=Test`. Als Suchparameter stehen Optionen zum Sortieren, Limitieren, Angaben ob gefundene Dokumente mit den Resultaten ausgegeben werden sollen oder Optionen, zum Bestimmen der maximalen Dauer der Suche zur Verfügung. Suchresultate werden in JSON Syntax dargestellt und geben Auskunft über Dauer der Suchanfragen, Anzahl der gefundenen Treffer, UUIDs und Score der Dokumente, in denen der gesuchte Begriff gefunden wurde und optional die Werte des Attributs selbst. [24] [4]

# 8. MIME in Haskell

Es existieren einige Haskell Bibliotheken die MIME Nachrichten behandeln. Eine vollständige Bibliothek, die alle notwendigen Funktionen im Umgang mit MIME Nachrichten bietet, ist nicht verfügbar. HaskellNet und die MIME Strike Force hat sich als Ziel gesetzt, einheitliche Bibliotheken zu erstellen bzw. zu definieren. Folgende Anforderungen wurden festgelegt: [29], [5]

**Lazy decoding** : Partielles Extrahieren und Modifizieren von relevanten Informationen aus MIME-Zeichenketten. In existierenden Bibliotheken muss die gesamte Zeichenkette geparst werden.

**Fehlertolerantes Parsing** von ungültigen MIME Nachrichten.

**Validierung** von MIME Zeichenketten auf fehlende oder erforderliche Kopfzeilen, Zeilen-Längen Beschränkungen, Zeichenkodierungen und anderen MIME Limitierungen.

**Transparenz:** Eine für den Entwickler transparent anwendbare Bibliothek, auch ohne spezielle Kenntnisse der zugrunde liegenden RFCs.

Trotz der Bemühungen ist es noch nicht gelungen eine solche Bibliothek zu schaffen. Im Prototyp werden für unterschiedliche Anforderungen geignete Funktionen aus folgenden Modulen verwendet:

**mime-string (Codec.MIME.String):** Diese Bibliothek hat seine Stärke im Parsen von MIME Nachrichten, spezifiziert in RFC 2045-2049 (siehe Kapitel 14.2).

**Text.Mime (HaskellNet):** Wird für die Komposition von MIME Nachrichten verwendet (siehe Kapitel 15.2).

# 9. GTK+ und Gtk2Hs

GTK+ ist ein plattformübergreifendes Toolkit, implementiert in C, zur Gestaltung von grafischen Benutzeroberflächen. Gtk2Hs ist ein Toolkit für Haskell welches Funktionen bereit stellt, mit dem GUI-Elemente manipuliert werden können. Die GTK+ Routinen beruhen auf prozeduraler Programmierung, so entsprechen auch die Gtk2Hs Funktionen direkten, imperativen Wrapper um die GTK+ Routinen. Im Haskell- Quellcode finden somit alle Manipulationen der GUI-Elemente in Do-Blöcken statt.

Die Hauptbestandteile der Oberfläche in GTK+ sind *Widgets*. Ein Widget ist ein Element einer grafischen Benutzeroberfläche, zum Beispiel ein Fenster, eine Schaltfläche oder

ein Textfeld. Diese Elemente werden in einem Baum gespeichert - das Widget “Fenster” entspricht dem Wurzelknoten des Baums - und bilden somit die Benutzeroberfläche. Mit dem GUI-Designer Glade werden verschiedene GTK+ Elemente erstellt, angeordnet und in einer XML-Datei gespeichert. Mit Hilfe des Gtk2Hs Toolkits wird diese Datei beim Applikationsstart geladen, die GTK+ Elemente in einem algebraischen Datentyp gespeichert und die Benutzeroberfläche dargestellt. Damit können Eigenschaften, das Erscheinungsbild und das Verhalten der GTK-Elemente auch während der Laufzeit des Programms manipuliert werden [23].

Bei GTK+ handelt es sich, wie bei vielen anderen GUI-Toolkits auch, um ein ereignisorientiertes Toolkit. Das Eintreten von Ereignissen löst Aktionen aus. Aktionen werden in Funktionen definiert und Gtk2Hs übergeben. Tritt nun ein bestimmtes Ereignis ein wird eine sogenannte “Callback Function” ausgeführt. Ein Beispiel für Ereignis und Aktion ist die Aktivierung einer Schaltfläche und die darauf folgenden Aktionen in der Benutzeroberfläche.

Der Kern eines GTK+ Programms ist die Hauptschleife. Hier wartet das Programm auf Aktionen des Benutzers und führt diese aus. Aufgaben müssen als separate Prozesse gestartet werden, damit die Benutzeroberfläche ansprechbar bleibt. GTK+ ist verantwortlich für die Abarbeitung der Hauptschleife; diese wird mit *initGUI* initialisiert und mit *mainGUI* gestartet [26], [3].

# Teil III.

# Realisierung des Prototyps

## 10. Funktionale Anforderungen

Es wurde ein Prototyp eines Mail-Clients als Haskell-Applikation mit GTK+ Oberfläche erstellt. Die Applikation namens *mailcdb* bietet eingeschränkt Unterstützung für die Verwendung in der Befehlszeile. In der dokumentorientierte Datenbank Apache CouchDB werden Maildaten sowie POP3 Mailserver-Zugangsdaten gespeichert. Die Applikation ermöglicht den Benutzer Dateien, die den MBOX Standard entsprechen (siehe Kapitel 2) in die CouchDB zu importieren. Mail-Nachrichten können von beliebig vielen POP3-Server bezogen werden. Die Anzahl an heruntergeladenen E-Mails pro Server wird protokolliert und die Nachrichten in die Datenbank übertragen. Zur Verwaltung der Mail-Nachrichten wurden Labels im Prototypen eingeführt. Jeder importierten oder erstellten Mail-Nachricht werden Labels zugewiesen, nach denen die Anzeige gefiltert werden kann.

Zu jeder Multipart MIME Mail-Nachricht werden Metainformationen der einzelnen MIME-Parts und der Anhänge angezeigt. Im Prototypen können Mail-Nachrichten mit Klartext-Kontent angezeigt werden. Für alle weiteren Medientypen wird die URL des CouchDB Anhangs zur Betrachtung im Webbrowser angezeigt. Das Versenden von einfachen Mail-Nachrichten mit einem MIME-Part, ist über einen in der Konfigurationsdatei angegebenen SMTP Server möglich. Bei der Speicherung und dem Versand von Mail-Nachrichten hält der Prototyp die in Kapitel 1.5 beschriebenen Mindestanforderungen an MIME Mail Clients ein, und erfüllt damit die RFC2049 MIME-Konformität.

Um eine effiziente Indexierung der Datenbank und eine hohe Performance bei Suchanfragen zu gewährleisten, wird Apache Lucene eingesetzt, dessen Implementation in Kapitel

7 und 16.4 beschrieben wird.

Im folgenden Kapitel wird die Repräsentation eines E-Mail Objekts in CouchDB beschrieben. Danach wird die Architektur meines Prototyps ausführlich beschrieben. Es werden verwendete, bestehende Module und Implementierungen neuer Module und Submodule vorgestellt.

# 11. E-Mail Nachrichten in CouchDB

Wie in Kapitel 6.1 beschrieben sind CouchDB Dokumente in sich geschlossene Einheiten von Informationen. Sie erzeugen die erste Ebene an Abstraktion über primitive Datentypen wie Integerwerten oder Zeichenketten, verleihen ihnen Struktur und logische Gruppierung und bilden somit die Entitätstypen.

Die wichtigste Datenstruktur im Prototypen sind E-Mail Nachrichten. Als Standard-Format von E-Mail hat sich die MIME-Spezifikation etabliert die in Kapitel 1 beschrieben wurde. Die in dem RFC beschriebene Menge an Kopfzeilen einer E-Mail wurde übernommen und entspricht den Attributen eines CouchDB Dokuments, welches im Prototyp eine Mail-Nachricht repräsentiert. Tabelle 2 stellt solch ein CouchDB Dokument dar und gibt zu jeder Kopfzeile ein Beispiel:

Bei *_id* und *_rev* handelt es sich um die Universally-Unified-ID und der Revisions-ID des CouchDB Dokuments. Mit Hilfe des Attributs *h_label* können abgelegte E-Mails kategorisiert und gekennzeichnet werden. Der Typ des Attributs ist eine Liste von Zeichenketten. Ein Mail kann somit mehrere Labels erhalten. Liegt ein E-Mail mit Klartext-Kontent vor, enthält das Attribut *h_body* den Nachrichtenkontent. Handelt es sich um ein E-Mail mit einem anderen Medientypen wird der binäre Kontent dem CouchDB Dokument als Anhang hinzugefügt und dessen Metadaten unter dem reservierten Attributnamen *_attachments* abgelegt. Auch Mailparts einer Multipart-Nachricht werden als CouchDB Anhänge gespeichert. *h_contentType* und *h_contentTypeParameter* geben die Kodierungsform im

| Attributname | Beispiel-Inhalt | Beschreibung |
|---|---|---|
| _id | 01f1f39ca32.. | UUID des E-Mails |
| _rev | 8-a6307b252.. | Revision ID |
| _attachments | | Anhänge und Mailparts |
| h_from | mike@example.com | Adresse des Senders |
| h_body | Hello, .. | Inhalt des Nachrichtenkörpers |
| h_to | joe@example.com | Adresse des Empfängers |
| h_cc | joe@example.org | carbon copy Adresse |
| h_label | [”Sent“] | Kategorisierung |
| h_sender | mike@example.com | Anzeigename des Senders |
| h_date | 2012/02/27 17:00 | Versanddatum |
| h_mime | 1.0 | MIME Version |
| h_subject | Meeting | Betreff |
| h_contentType | text/html; | Kontenttyp des Nachrichtenkörpers |
| h_contentTypeParameter | charset=ISO-8859-15 | Parameter des Kontenttyp |

**Tabelle 2:** Dokument Attribute eines Mailparts in CouchDB

ursprünglichen MIME-Mail an; alle weiteren Attribute sind selbst erklärend.

# 12. Architektur

Um einen geringen Grad der Kopplung und eine funktionelle Trennung zwischen den Modulen zu erreichen, habe ich mich auf die klassische “Drei-Schichten-Architektur” gestützt, bei der Präsentation, Logik und Datenhaltung voneinander im Quellcode getrennt und gekapselt sind. Zugriffe erfolgen jeweils nur auf direkt benachbarte Schichten. In Abbildung 2 wird der schematische Aufbau der Module des Prototyps dargestellt.

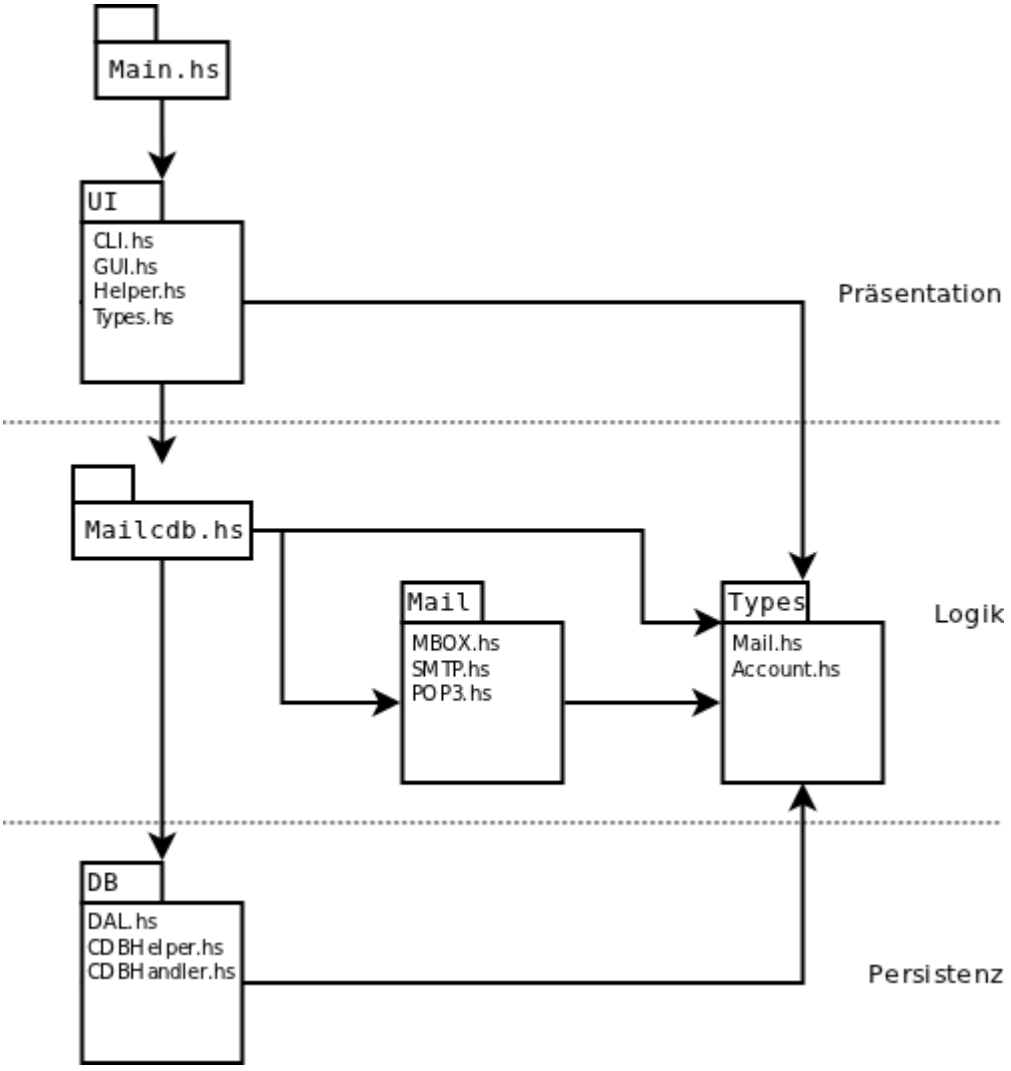


**Abbildung 2:** Modulkopplung

Das Modul *Main.hs* stellt die Funktionen für Einstiegspunkt in das Programm zur Verfügung und startet abhängig von den angegebenen Parameter die grafische oder Befehlszeilen-Variante des Prototypen.

Im Modul *UI* befinden sich die Submodule CLI und GUI. Funktionen für die Anwendung von mailcdb über die Kommandozeile sind in CLI (Command Line Interface) definiert. Das Submodul GUI (Graphical User Interface) ist verantwortlich für die Darstellung der mit Hilfe von GTK+ erstellten grafischen Schnittstelle und der Implementation deren Funktionen. Implementationen von Event-Handlern, Funktionen die GUI-Elemente manipulieren und weitere Aktionen auf gedrückte Schaltflächen sind im Submodul

Helper enthalten (siehe Kapitel 13).

Das Modul *Mailcdb.hs* beinhaltet alle Top-Level Funktionsaufrufe des Prototypen:

**runImportMboxFrom:** Importiert Mail-Nachrichten aus einer MBOX Datei in die CouchDB.

**runRetrieveMailFromActivePopAccounts:** Lädt Mail-Nachrichten von POP-Server herunter.

**runNewMail:** Erstellt eine neue Mail-Nachricht und speichert diese in der CouchDB.

**runNewMailReply:** Erstellt eine Mail-Antwort und speichert diese in der CouchDB.

**runNewMailForward:** Erstellt eine Mail-Weiterleitung und speichert diese in der CouchDB.

**runSendMail:** Sendet ein bestehendes Mail über einen SMTP-Server.

**runDeleteMail:** Löscht ein Mail in der CouchDB.

Funktionen, um Mail-Nachrichten zu importieren oder zu versenden, werden im Modul *Mail* zusammengefasst. Das Submodul *MBOX* implementiert Funktionen zum Öffnen und Importieren von Nachrichten aus MBOX-Datenformaten. Submodul POP3 verbindet mit POP3 Mailservern und lädt neue Mail-Nachrichten herunter und Submodul *SMTP* versendet MIME-Mails über das Transferprotokoll SMTP (siehe Kapitel 15).

In Modul *Types* werden intern verwendete Datentypen und Transformationsfunktionen in den Submodulen *Account* und *Mail* definiert (siehe Kapitel 14).

Das Modul *DB* ist verantwortlich für die Speicherung von Daten im CouchDB Datenbanksystem. *CDBHandler* beinhaltet die Implementierung von Funktionen, um neue

| Haskell Package | Modul | Version | Entwickler |
|---|---|---|---|
| Mailcdb | DB.CDBHandler | 1.0 | Karl Beranek |
| Mailcdb | DB.CDBHelper | 1.0 | Karl Beranek |
| Mailcdb | DB.DAL | 1.0 | Karl Beranek |
| Mailcdb | Mail.MBOX | 1.0 | Karl Beranek |
| Mailcdb | Mail.POP3 | 1.0 | Karl Beranek |
| Mailcdb | Mail.SMTP | 1.0 | Karl Beranek |
| Mailcdb | Mailcdb | 1.0 | Karl Beranek |
| Mailcdb | Main | 1.0 | Karl Beranek |
| Mailcdb | Types.Account | 1.0 | Karl Beranek |
| Mailcdb | Types.Mail | 1.0 | Karl Beranek |
| Mailcdb | UI.CLI | 1.0 | Karl Beranek |
| Mailcdb | UI.GUI | 1.0 | Karl Beranek |
| Mailcdb | UI.Helper | 1.0 | Karl Beranek |
| Mailcdb | UI.Types | 1.0 | Karl Beranek |
| CouchDB | UI.Types | 1.2 | Arjun Guha |

**Tabelle 3:** Module

Dokumente in CouchDB zu erstellen, aufzurufen und zu löschen sowie CouchDB Views und Lucene Indezes abzufragen. Mit Funktionen in CDBHelper wird die Datenbank für die Verwendung des Prototyps vorbereitet. In Zuge dessen, werden Views und deren Map- und Reduce-Funktionen sowie Indexfunktionen für die Volltextsuche erstellt. Das Modul *DBAL* fungiert als Abstraktionsschicht, welche die typsicheren Funktionen zum Zugriff auf die Datenbank beinhaltet (siehe Kapitel 16).

Tabelle 3 führt alle Module des programmierten Haskell Package *mailcdb* sowie existierende, modifizierte Haskell Module an.

# 13. UI - User Interface

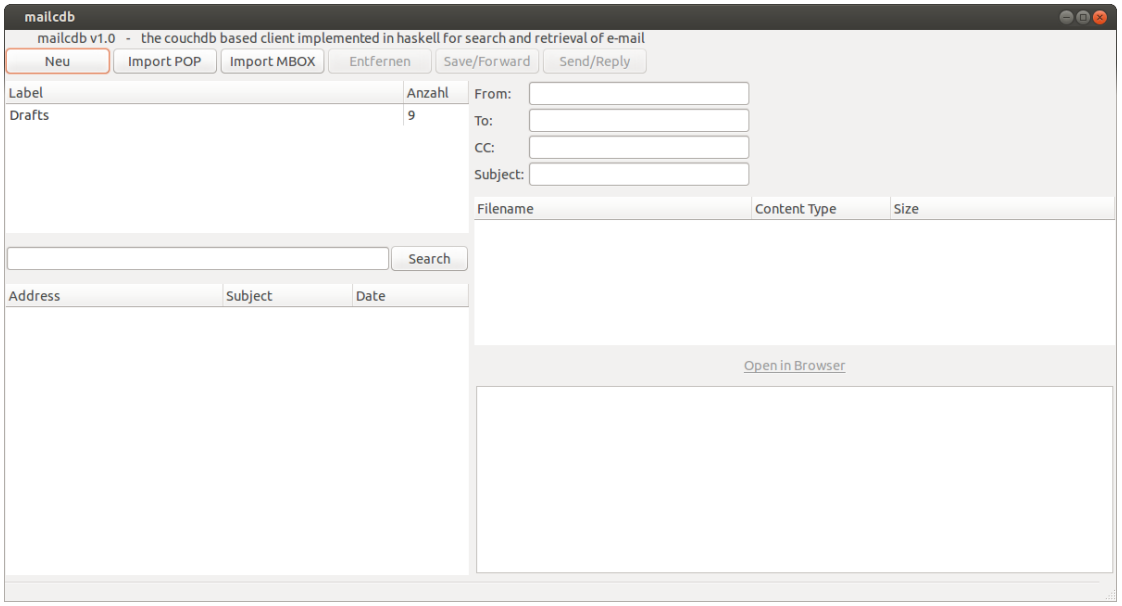


**Abbildung 3:** mailcdb Screenshot

Die Benutzeroberfläche des Prototypen ist in drei Bereiche eingeteilt: ein Listenfeld mit Labels, ein Listenfeld mit Mail-Nachrichten und die Detailansicht von einzelnen E-Mails. Die Liste von Labels zeigt alle in der Datenbank verfügbaren Labels mit der Anzahl der vorkommenden Nachrichten an. Selektiert der Benutzer ein Label, werden Nachrichten mit gleichen Label in der Mail-Liste angezeigt. Nach der Auswahl einer Nachricht können nun verschiedene Funktionen ausgeführt werden, abhängig davon welcher Label ausgewählt wurde. Nachrichten mit Label “Drafts” oder “Outbox” können bearbeitet, gespeichert oder versendet werden. Für Nachrichten mit allen anderen Labels können Antworten oder Weiterleitungen erstellt werden. Die Abbildung 4 stellt diesen Sachverhalt in einem Ablaufdiagramm dar. Des Weiteren steht ein Eingabefeld für die Suche über das integrierte Apache Lucene Framework zur Verfügung.

Statische Elemente der Benutzeroberfläche wurden mit dem GUI-Designer Glade angeordnet und in gui.glade gespeichert. Viele GUI Elemente werden während der Laufzeit durch Programmcode manipuliert. Dazu zählen vor allem Inhalts-Attribute von Text- und Listenfelder, in denen Labels, Mail-Header und Mail-Inhalte anzeigt werden. Auch die Aktivierung von Schaltflächen und URLs wird in Abhängigkeit des gewählten La-

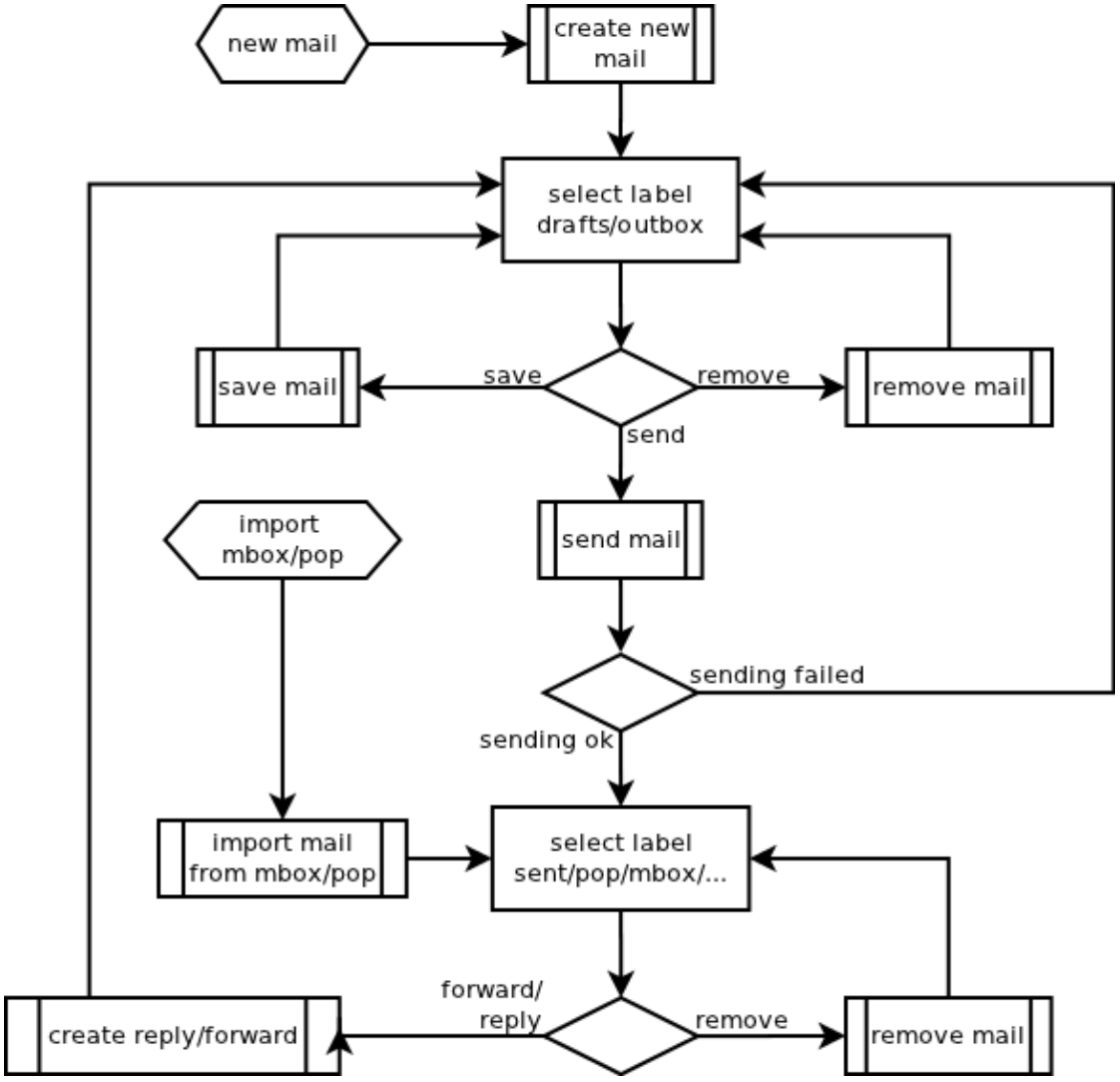


**Abbildung 4:** Ablaufdiagramm von Funktionen in Abhängigkeit zum ausgewählten Label

bels durch Programmcode geregelt. Die Dokumentation des Quellcodes (siehe Anhang **??**) führt alle grafischen Elemente und deren Verwendungszweck an die während der Programmausführung manipuliert werden.

## 13.1. Modul GUI: Graphical User Interface

Die grafische Benutzeroberfläche des Prototyps wird in diesem Modul implementiert. In *GUI.hs* werden alle Elemente der Oberfläche geladen, mit entsprechenden Parametern konstruiert und Ereignisfunktionen zu Schaltflächen und Listenfelder zugewiesen. Alle

Ereignisfunktionen und Funktionen, die gewisse Abläufe wie das Senden von Mail-Nachrichten oder das Erstellen eines leeren, neuen Mail-Nachricht steuern, befinden sich im Submodul *Helper.hs* sowie diverse Hilfsfunktionen von GUI Elementen. Diese Elemente werden mittels des Gtk2Hs Moduls realisiert, welches eine umfangreiche Schnittstelle zu der GTK+ Bibliothek zur Verfügung stellt (siehe Kapitel 9). *Types.hs* beinhaltet Definitionen von Datentypen verwendet im UI Modul.

## 13.2. Modul CLI: Command Line Interface

Das CLI Modul stellt Basis-Funktionalitäten, für die Verwendung des Prototypen in einer Kommandozeile bereit. Die beim Aufruf der Anwendung verwendeten Parameter werden in *runCLI* geparst und entsprechenden Funktionen übergeben. Es können Mailkonten angelegt und verwaltet werden, das Abrufen von Nachrichten von POP3-Servern angestoßen oder Importe von MBOX Dateien gestartet werden. Der Aufruf ohne Parameter oder mit dem Parameter “start” startet die graphische Benutzeroberfläche des Prototypen.

# 14. Types

In diesem Modul werden wesentliche, benutzerdefinierte Datentypen definiert, die von anderen Modulen im Prototyp verwendet werden.

## 14.1. Modul Account

Dieses Submodul führt den algebraischen Datentyp *Account* mit dem Konstruktur *POP3Account* ein. Dieser Typ beinhaltet alle Information, die nötig sind um eine Verbindung zu einem POP3 Server aufzubauen, einen Benutzer zu authentifizieren sowie die

| Komponente | Type | Beschreibung |
|---|---|---|
| p_host | String | POP3 Server Name |
| p_user | String | Benutzername |
| p_pass | String | Kennwort |
| p_email | String | E-Mail Adresse |
| p_label | String | Name des Labels |
| p_countRetr | Int | Anzahl heruntergeladener E-Mails |
| p_enabled | Bool | aktiviert/deaktiviert |

**Tabelle 4:** Komponenten des Typs Account

Anzahl heruntergeladener E-Mails, Name des Labels und einen boolsches Feld für die Aktivierung bzw. Deaktivierung des Mail-Kontos (siehe Tabelle 4).

## 14.2. Modul Mail

Das Modul Mail definiert den Datentyp *mailcdb:Mail* bestehend aus dem Datentyp *mailcdb:Header* und einer Liste von Elemente des Typen *mailcdb:Attachment*. *mailcdb:Header* ist ein algebraischer Datentyp mit Komponenten, die den Attributen eines Mail CouchDB-Dokuments entsprechen (siehe Kapitel 11) und die Kopfzeilen einer Mail-Nachricht speichert. *mailcdb:Attachment* hält den Namen, die Größe, den Kontenttyp und den in Base64 kodierten Kontent von Anhängen und Mailparts bereit. Damit repräsentiert der Datentyp *mailcdb:Mail* das Informationsobjekt für eine Mail-Nachricht im Prototypen (siehe Tabelle 5).

Des Weiteren wird ein Typ MailInfo erstellt, welches nur die wesentlichsten Informationen von Mail-Nachrichten wie Email-Empfänger oder Sender, Datum und Betreff beinhaltet und für Elemente im GUI verwendet wird. Beide Typen gewährleisten Zugriffe auf die aus der CouchDB geladenen Entitäten, über die durch die Record-Syntax implizit gegebenen Zugriffsfunktionen.

| Komponente | Type | Beschreibung |
|---|---|---|
| a_filename | String | Dateiname des Anhangs |
| a_type | String | Kontenttyp |
| a_content | String | Kontent |
| a_length | Int | Anzahl Zeichen des Kontents |

**Tabelle 5:** Komponenten des Typs Attachment

Eine wichtige Funktion in diesem Modul ist die Funktion *convertMessageToMail*. Dieser Funktion wird als Argument eine MIME-Nachricht vom Typ *mime-string-0.4:Message* übergeben, welche in den von Prototypen primär verwendeten E-Mail Typ *mailcdb:Mail* konvertiert wird. MIME-Nachrichten werden in den Modulen *Mail.POP3* und *Mail.MBOX* erzeugt, wie in Kapitel 15.1 und 15.3 beschrieben.

**Typ: mime-string-0.4:Message** Dieser Typ wird im Paket *mime-string* implementiert und erlaubt eine Dekomposition eines Mails im MIME-Datenformat in dessen Bestandteile. Diese Präsentation einer MIME-Mail ist strukturiert und speichert Felder redundant. So werden die Kopfzeilen-Informationen einer E-Mail zusätzlich zum geparsten Kontent auch als Klartext, sowie als Namen-/Wertepaare gespeichert. Multipart-Mails werden mittels rekursiven Typen realisiert. [10]

**Typ: mailcdb:Mail** Durch die Konvertierung werden zur Weiterbearbeitung der MIME-Entität notwendige Informationen extrahiert und mit dem Datentyp *mailcdb:mail* präsentiert. Die Funktion *convertMessageToMail* bedient sich der Funktionen *parseContent*, *parseInfo* und *parseAttachment* die die Datentypen *MessageContent* und *MessageInfo* parsen und Nachrichtenkontent bzw. Kopfzeilen-Informationen extrahieren. Relevante Daten von Kopfzeilen und Klartext-Kontent werden als *mailcdb:Header* gespeichert. Handelt es sich beim Medientyp der MIME-Entität um einen anderen Typen als text/plain, wird dieser Base64-kodiert und als *mailcdb:Attachment* hinzugefügt. Auch Mailparts

von Multipart-MIME-Mails werden Base64-kodiert und als *mailcdb:Attachment* gespeichert.

# 15. Mail

Die Submodule im Modul *Mail* beinhalten Funktionen um externe Datenquellen anzubinden und Mail-Nachrichten zu importieren sowie E-Mails an Mail-Server zu übermitteln. Mit dem POP3 Konnektor (Modul *Mail.POP3*) werden über das POP3 Übertragungsprotokoll Nachrichten von Mailservern abgerufen. Nachrichten die im MBOX-Dateiformat gespeichert sind, werden über das Modul *Mail.MBOX* importiert und ausgehende E-Mails werden über Funktionen im Modul *Mail.SMTP* gesendet.

## 15.1. Modul POP3

Das Übertragungsprotokoll POP3 (*Post Office Protokol Version 3*) wird verwendet um Nachrichten von einem Mailserver abzuholen. In diesem Modul werden alle Funktionen, die den Zugriff auf einen POP3 Mailserver benötigen, implementiert. Der Vorgang ist im folgenden beschrieben [22]:die

1. die Verbindung zum POP3 Server wird geöffnet,
2. die Autorisierung mit gültigen Benutzerdaten erfolgt,
3. die Differenzmenge an neuen E-Mails wird ermittelt,
4. neue E-Mails werden heruntergeladenen und die Anzahl der heruntergeladenen E-Mails angeglichen,
5. die Verbindung zum POP3 Server wird getrennt;

**Funktion getNewMessages** Diese Funktion erwartet einen Parameter vom Typ *Account* und returniert eine Liste von Nachrichten vom Typ *mime-string-0.4:Message*. Der Datentyp *POP3Handler* wird mit der Funktion *connectToPopServerWith* konstruiert und die für die Kommunikation mit dem POP3-Server erforderlichen IO-Funktionen aufgerufen. Die Menge an bisher nicht heruntergeladenen Mail-Nachrichten wird ermittelt und vom Mailserver heruntergeladen.

**Funktion retrieveMailFromAccount** Diese Funktion bedient sich der Funktion getNewMessages, um eine Liste von Nachrichten des Typs *mime-string-0.4:Message* von einem POP3 Server zu holen und diese in eine Liste von Elementen des Typs *mailcdb:Mail* zu konvertieren. Im Fehlerfall fängt die Funktion IO-Exceptions der Funktion *getNewMessages* und returniert den Datentyp Left mit entsprechender Fehlermeldung.

**Datentyp POP3Handler** Der Record-Datentyp *POP3Handler* kapselt den IO-Code des POP3 Handlers in einer funktionalen Datenstruktur, damit er in puren Funktionen verwendet werden kann. Die Funktion *connectToPopServerWith* dient dabei als Konstruktur, der die Implementierungen mit IO-Code enhält. Diese Struktur ermöglicht die IO Funktionen als Parameter zu verwenden, als Resultate zu übergeben oder partiell zu evaluieren.

**Funktion connectToPopServerWith** Diese Funktion enthalten den IO-Codesegmente, die für die Kommunikation mit einem POP3-Server benötigt werden. Eine minimale Verbindung eines Clients, mit einem POP3-Server, besteht aus dem Verbindungsaufbau, der Autorisierung, eines Befehls zum Abholen oder Abfragen von E-Mail Nachrichten auf den Server und der Trennung der Verbindung. [22] Diese werden mit den IO Funktionen *ph-conn*, *ph-fetchNew* und *ph-disconn* implementiert.

**Funktion HaskellNet.POP3** Die protokoll-nahe POP3-Kommunikation mit dem Server wird über das importierte Modul *Network.HaskellNet.POP3* bewerkstelligt. Das Modul ist Teil des HaskellNet Pakets und bietet für alle in der RFC1939 [22] definierten Operationen wie *user* und *pass* (Autorisierung), *retr* (Abholen einer Nachricht) oder *quit* (Trennung einer Verbindung) entsprechende Funktionen. Beim Verbindungsaufbau mit Funktion *connectPop3* wird ein Handle vom Typ *IO (POP3Connection Handle)* zurückgegeben, welcher den folgenden Funktionen als Parameter übergeben wird. Die Funktion *retr* erwartet die fortlaufende Nummer einer Mail-Nachricht auf dem POP3 Server und gibt die empfangene Nachricht als *IO (Byte-String)* zurück.

## 15.2. Modul SMTP

Für das Senden von E-Mails ist das Modul SMTP verantwortlich. Es werden Plaintext-Mails über einen in der Konfigurationsdatei angegebenen SMTP-Server versendet. Versenden von E-Mails mit Anhängen wird in der Funktion *doSendMimeMail* unterstützt, aber im Prototypen nicht verwendet.

**doSendMimeMail** Diese Funktion ermöglicht das Senden von MIME-Mails. Die Funktion erwartet den zu sendenen Mailheader, den Pfad einer anzuhängenden Datei, Hostname des SMTP-Servers sowie die zu Senderadresse. Der übergebene Mailheader wird zu einem MIME Objekt transformiert. Für Zeichen außerhalb des ASCII Zeichensatzes in Betreff oder Kontent des E-Mails wird die Quoted-Printable-Kodierung angewendet. Die versendete Mailnachricht enthält den Kontent als Plain-Text und einen Mailpart des Typs *HTML-Message*.

**HaskellNet: SMTP** Dieses Modul stellt die Funktionalität der protokollnahen Kommunikation, um Mail-Nachrichten über einem SMTP Server zu versenden, zur Verfügung. Eine ähnliche Funktionalität wird mit dem Modul SMTPClient [25] erreicht, jedoch

konstruiert dieses Modul einen fehlerhaften SMTP-Envelope um die MIME-Nachricht. Daher wird die Implementation von HaskellNet verwendet.

Funktionen wie *connectSMTP* und *closeSMTP* übernehmen die SMTP Kommunikation mit dem Server und ermöglichen Verbindungsaufbau und -trennung. Die Funktion *sendMimeMail* ermöglicht das Senden von MIME konformen E-Mails und erwartet Sender- und Empfänger-Mailadresse sowie den E-Mailkontent als Byte-String. Fehler während des Sendens lösen eine IO-Exception aus.

## 15.3. Modul MBOX

**doImportMbox** Die Funktion öffnet den per Parameter übergebenen Dateipfad einer MBOX Datei, trennt die Zeichenketten unterschiedlicher Mail-Nachrichten und parst die Nachrichten mit der Funktion *mime-string:parse*. Die Funktion returniert den Typ *mime-string:Message*, der auch beim Import von Nachrichten über das POP3 Protokoll verwendet wird. Die Liste dieser Werte wird konvertiert zu einer Liste vom Typ *mailcdb:Mail* und bei fehlerfreier Ausführung als Either-Right Konstrukt zurückgegeben.

**MBOX Parsing** Zum Parsen der mbox Datei wurde das Modul *Codec.Mbox* sowie *Data.MBox* [1] getestet. Erst genanntes Modul arbeitet mit Lazy Byte-Strings und bietet eine effizientere Verarbeitung der MBOX Datei. Es extrahiert allerdings nur die Kopfzeilen-Informationen Absender und Empfangsdatum der Nachrichten. Funktionen in dem Modul *Data.MBox* parsen die MBOX Datei und bilden diese in einem granular aufgebauten Objekt vom Typ *Message* ab, in dem auch alle Kopfzeilen-Informationen berücksichtigt werden. Weitere Tests mit diesem Modul zeigten auf, dass keine Multipart-Nachrichten korrekt identifiziert und geparst werden konnten. Das bestgeeignete Parsing, welches geforderte Funktionaliten bietet, wird vom Modul *mime-string* zur Verfügung gestellt. [10]

# 16. DB

In diesem Modul sind Funktionen definiert, die den Zugriff auf Dokumente in der CouchDB und die Erstellung von Design Dokumenten für Views, Indexfunktionen und Validierung zur Verfügung stellen.

## 16.1. Modul CDBHandler

In diesem Modul werden Funktionen implementiert, die die Manipulation von CouchDB-Dokumenten ermöglichen. Dokumente werden erstellt mit *storeDocument*, geladen mit *retrieveDocument*, aktualisiert mit *updateDocument* und gelöscht mit *deleteDocument*. Als Parameter übergeben und identifiziert werden Dokumente jeweils anhand der ID-Zeichenkette, die der UUID des Dokuments in der Datenbank entspricht. Rückgabewerte von Abfragen sind die UUID und ein Wert der Typklasse Data, welches das als JSON übergebene Dokumente darstellt. Durch die Monomorphismus Einschränkung wird der Typ in dem das Dokument konvertiert wird in der aufrufenden Funktion in der Typ Signatur festgelegt. Im Fehlerfall, Probleme bei Abfrage der CouchDB oder bei der JSON-Konvertierung wird jeweils ein *Left* Wert vom Typ *Either* mit entsprechender Fehlermeldung returniert.

Für die Abfragen von Views stehen zwei Funktionen zur Verfügung (*retrieveDocFromView*, *retrieveIdDocFromView*) welche als Parameter den Datenbanknamen, den Namen des CouchDB Views und die Abfrage Parameter akzeptieren. Die erste Funktion liefert die vom View erzeugten Dokumente zurück, die Funktion *retrieveIdDocFromView* beinhaltet im Resultat zusätzlich die UUIDs.

Um den von CouchDB-Lucene erzeugten Index abzufragen, werden die Funktionen *queryDB* und *queryDBKeys* verwendet, die mit den gleichen Parametern, wie zur Abfrage von Views, anzusprechen sind.

## 16.2. Modul CDBHelper

Die Funktionen in Modul CDBHelper erstellen die für den Prototypen benötigten CouchDB Design Dokumente. Die Java Script Funktionen von Map- und Reduce-Funktionen werden in den Design Dokumenten gespeichert und bilden die während der Laufzeit des Prototypen benötigten CouchDB Views. Auch die Indexfunktion für die Volltextsuche mittels Apache Lucene, sowie die Validierungsfunktionen für gültige CouchDB Dokumente werden in den Design Dokumenten *_design/search* und *_design/validate* definiert (siehe Kapitel 16.4).

## 16.3. Modul DAL (Data Access Layer)

Das Modul konvertiert die Funktionen aus *CDBHandler* in eine typsichere Version um Typensicherheit zu nutzen. Für Datenbank Manipulationen bietet das Modul *CDBHandler* Funktionen, die als Parameter Typinstanzen der Typklasse Data akzeptieren. Funktionen im Modul *DAL* bestimmen den Typ durch die explizite Typ Signatur. Zum Beispiel die in *CDBHandler.hs* generisch definierte Funktion *storeDocument*, wird von den typsicheren Funktionen *storeMail* und *storeAccount* in *DAL.hs* verwendet, die nur den Typ *mailcdb:Mail* bzw. *mailcdb:Account* akzeptieren.

**Funktion storeMail** Eine wichtige Funktion dieses Moduls ist die Funktion *storeMail*. Sie erwartet einen Parameter des Typs *mailcdb:Mail*, siehe Kapitel 15.1.

Beinhaltet das Mail eine leere Liste von *mailcdb:Attachments*, sind also keine Anhänge zu der Mail-Nachricht existent, wird die IO Funktion zum Speichern von CouchDB Dokumenten von CDB-Handler aufgerufen. Existieren ein oder mehrere Anhänge zu der Mail-Nachricht, wird aus den Headerinformationen und den Anhängen ein JSON Objekt konstruiert und mit IO Funktionen von CDB-Handler ein CouchDB Dokument mit *Inline Attachments* in die Datenbank geschrieben.

| Design Dokument | View | Beschreibung |
|---|---|---|
| _design/accounts | byLabel | Konten gruppiert nach Label |
| _design/accounts | countRetr | Anzahl der heruntergeladenen Nachrichten jedes Kontos |
| _design/accounts | enabled | aktivierte Konten |
| _design/mails | attachments | Anhänge gruppiert nach UUID |
| _design/mails | incoming-byLabel | MailInfo gruppiert nach Label (Attribut mAddress entspricht From) |
| _design/mails | labels | Labels und Anzahl von Nachrichten pro Label |
| _design/mails | mailInfo | MailInfo |
| _design/mails | outgoing-byLabel | MailInfo gruppiert nach Label (Attribut mAddress entspricht To) |

**Tabelle 6:** CouchDB Views

Die CouchDB Views, angeführt in Tabelle 6, werden in den Design Dokumenten *_design/mails* und *_design/accounts* für den Abruf von Daten aus der CouchDB definiert.

## 16.4. CouchDB Design Dokumente

**Indexfunktion** In der Indexfunktion werden die zu indizierenden Attribute von CouchDB Dokumenten angegeben. Tabelle 7 zeigt die indexierten Attribute und verwendeten Index Optionen. Da der Informationsgehalt des Attributs “Subject” am größten ist, wird dieser durch den höheren Boost Wert in den Suchresultaten priorisiert.

**Validierung** Der Prototyp verwendet zwei Typen von CouchDB Dokumenten: *Mail* und *Account*. Er bedient sich des Konzepts des Duck Typing, um Typen zu identifizieren

| Attribut | Index Optionen | Boost Wert |
|---|---|---|
| Body | UnStored | 1 |
| CC | UnStored | 1 |
| Date | UnStored | 1 |
| From | Keyword | 1,2 |
| Subject | Keyword | 1,5 |
| To | Keyword | 1,2 |

**Tabelle 7:** Indizierte Attribute

und zu validieren. Dokumente des Typs *Mail* besitzen das nicht leere Attribut *h_from* und benötigen zur Validierung die nicht leeren Attribute *h_contentType* und *h_label*. Der Typ *Account* wird am Vorhandensein des nicht leeren Attributs *p_host* kenntlich gemacht und benötigt das nicht leere Attribut *p_label*. Der Javacode dieser Funktionen wird in dem Design Dokument *_design/validate* gespeichert. Durch diese simple Methode wird eine effektive Validierung erreicht und das Erstellen und Ändern von Dokumenten in der Datenbank, die diesen Anforderungen nicht genügen, verhindert.

Beide Typen von CouchDB Dokumente werden im Haskell Code in gleichnamige abstrakte Datentypen *Mail* und *POP3Account* konvertiert (siehe 14.2).

# 17. Database

## 17.1. Modul CouchDB

Die zum Zeitpunkt der Erstellung des Prototyps aktuelle Version des Moduls CouchDB, ist die Version 0.10.1. Diese Version unterstützt die Erstellung von Design-Dokumenten für CouchDB Views. CouchDB-Lucene benötigt Indexfunktionen, die in Design-Dokumenten gespeichert werden. Der syntaktische Aufbau dieser ist jedoch anders als der von Design-

Dokumenten für CouchDB Views. Auch die Validierung von CouchDB Dokumenten wird über in Design-Dokumenten abgelegten Funktionen ermöglicht. Dieser Typ von Design-Dokumenten kann in Version 0.10.1 nicht erstellt werden. Damit diese benötigten Design Dokumente vom Prototyp automatisch angelegt bzw. Suchresultate abgefragt werden können, wurde das Modul *CouchDB* in mailcdb importiert und mit den Funktionen *newDesignDoc*, *newFTI* und *queryFTI* im *CouchDB.hs* und *Unsafe.hs* erweitert.

**CouchDB Attachments** CouchDB Dokumente werden geparst als JSON Objekte an die CouchDB-Schnittstelle übergeben. Auch CouchDB Attachments müssen zu JSON Objekten geparst werden und in ein CouchDB Dokument eingebettetet werden. Handelt es sich um binäre Daten werden diese Base64 kodiert.

# 18. Fehlerbehandlung

Die größte Quelle von Fehlern existiert beim Umgang mit Ein- und Ausgabe Funktionen. In einigen solcher Funktionen mit Seiteneffekten wird direkt auf Dateien oder Netzwerk-Streams zugegriffen, welche unerwartete Rückgabewerte liefern könnten. Solche Fehler oder unerwartete Ereignisse werden mit den Funktionen *catch* und *try* aus dem *System.IO.Error* Modul gefangen. Die gefangenen Exceptions werden gekapselt in einem Either Datentyp an die aufrufende Funktion kommuniziert. Ein returnierter *Left*-Wert weist auf einen Fehler hin und entspricht den Fehlertext; ein *Right*-Wert entspricht dem gewünschten Rückgabewert bei fehlerfreier Ausführung der IO Monade.

# 19. Externe Module

Tabelle 8 listet alle Abhängigkeiten von mailcdb auf und beschreibt deren Verwendungszweck:

| Package | Verwendung |
|---|---|
| base | Prelude |
| base64-string | Base64 Kodierung von Anhängen |
| bytestring | Konvertierung von ByteStrings |
| ConfigFile | Einlesen von mailcdb Konfigurationsdatei |
| CouchDB | Schnittstelle mit CouchDB |
| glade | Import von Konfigurationsdatei für Benutzeroberfläche |
| gtk | Darstellung und Manipulation von grafischer Benutzeroberfläche |
| HTTP | Abhängigkeit von CouchDB |
| HaskellNet | Versenden von Mail über SMTP |
| json | Abhängigkeit von CouchDB |
| mime-string | Verarbeitung von MIME Typen |
| mtl | Abhängigkeit von CouchDB |
| network | Abhängigkeit von CouchDB |
| old-locale | Lesen von Regionseinstellungen |
| old-time | Lesen von Systemzeit |
| time | Formatierung von Uhrzeit in Mail-Nachrichten |
| text | Konvertierung von Unicode |
| utf8-string | UTF8-Kodierung |

**Tabelle 8:** Abhängigkeiten

# 20. Entwicklungs Umgebung

Der Prototyp wurde unter Linux Ubuntu 12.04 entwickelt. Als integrierte Entwicklungsumgebung ist Leksah 0.12.1.3 mit GHC 7.6.2 zum Einsatz gekommen. Tabelle 9 führt Versionen der verwendeten Software-Pakete an.

| Hersteller | Paket | Version |
|---|---|---|
| Apache | Apache CouchDB | 1.2 |
| Apache | Lucene | 1.2 |
| Glade | Glade | 3.8.0 |
| Haddock | Haddock | 2.13.2 |

**Tabelle 9:** Paketversionen

## 20.1. Dokumentation des Quellcodes

Für die Dokumentation des Quellcodes wurde Haddock verwendet. Haddock ist ein Tool für die automatische Generierung einer HTML-Dokumentation aus Haskell Quellcode Dateien, wobei auch Informationen von Typ-Definitionen extrahiert werden. Jedes Modul und alle Definition von Funktionen oder Datentypen sind im Haskell Quellcode entsprechend den Syntax-Regeln von Haddock kommentiert [11]. Ein PDF-Export der HTML-Dokumentation befindet sich in Anhang **??**.

# Teil IV.

# Resultate

## 21. Einleitung

In diesem Abschnitt wird die Performancemessung des Prototypen beschrieben und mit Messungen von etablierten Anwendungsprogrammmen die E-Mail Nachrichten speichern verglichen. Damit die Testergebnisse nicht von externen Faktoren wie z.B. Übertragungsgeschwindigkeit im Netzwerk, Antwortverhalten von POP3 Servern verfälscht werden, wurde der Import von Nachrichten aus einer MBOX-Datei untersucht. Ziel der Messung ist die Erhebung der Dauer des Importvorgangs und der Speicherbedarf in der neuen Speicherungsform.

## 22. Test Umgebung

In den Tests werden drei Mail-Clients verglichen. Es existiert eine große Anzahl an E-Mail Clients die das MBOX Datenformat unterstützen. Ein populärer E-Mail Client unter Linux ist die Software Gnome Evolution. Die Software speichert Maildaten direkt im MBOX Standard in einer vorgegebenen Verzeichnisstruktur.

Mit dieser Software wurde das Test-Sample erstellt: eine 29,7 MB große MBOX Datei, bestehend aus 1.000 Mail-Nachrichten, wovon rund 10 Prozent der Nachrichten binäre Anhänge besitzen. Die Ergebnisse des Imports der MBOX-Datei in Gnome Evolution stellt die Basislinie der Vergleichstests dar.

Als weiterer Vergleichskandidat kommt Microsoft Outlook 2007 zum Einsatz. Der Mail-

Client verwendet zur Speicherung das Format PST. Ein Import ist hier nur über zwei Zwischenschritte möglich, da Microsoft Outlook keinen direkten Import von MBOX unterstützt. Der erste Schritt ist die Konvertierung von MBOX auf EML und damit die Übernahme der E-Mails in Microsoft Outlook Express. Im zweiten Schritt werden Maildaten von Outlook Express nach Outlook importiert und ins eigene Format PST übernommen.

# 23. Bewertung

Tabelle 10 zeigt die gemessenen Zeiten im Durchschnitt über 5 wiederholte Messungen bei vernachlässigbarer Varianz. Die Messungen zeigen, dass der Import nach MS Outlook nur wenig mehr Zeit benötigt als Gnome Evolution und damit knapp über der Basisline liegt. Der Import beim Prototypen mailcdb benötigt rund ein Drittel der Zeit länger. Detailmessungen einzelner Funktionen des Prototypen zeigen, dass die Erstellung der CouchDB Anhänge zwei Drittel der Gesamtdauer benötigen. Da die CouchDB-Schnittstelle nur das Hochladen von CouchDB-Anhängen in Form von Inline Attachments unterstützt, müssen Mailparts Base64-kodiert und ins JSON Objekt inkludiert werden. Die Base64 Kodierung verursacht höhere CPU-Lasten und einen Zuwachs der Datenmenge um ein Drittel, welches demnach die Performance beim Import neuer Daten negativ beeinträchtigt.

Ein effizienteres Hochladen von binären Daten in CouchDB ist möglich bei Verwendung des *MIME multipart/related* Formats. Eine Implementierung der Unterstützung von *MIME multipart/related* in die CouchDB- Schnittstelle entspricht jedoch einer Neuentwicklung der Schnittstelle, da diese wesentlich auf das JSON Datenformat aufbaut.

Nach dem Import der Daten in CouchdB werden die binären Daten in CouchDB automatisch komprimiert. Der Speicherbedarf in CouchDB ist somit mit 24,8MB um ca. 16% geringer als die Speicherung im Dateisystem.

| Mail-Client | Dauer | Speicherbedarf |
|---|---|---|
| Gnome Evolution (Basisline) | 44 s | 29,7MB |
| MS Outlook | 52 s | 26,7MB |
| mailcdb | 75 s | 24,8MB |

**Tabelle 10:** Messungen

# 24. Schlussfolgerung

Ein Prototyp zur Speicherung von E-Maildaten, unter Verwendung der dokumentbasierten Datenbank CouchDB, wurde mit der Programmiersprache Haskell entwickelt. Ein GTK-Frontend ermöglicht, gespeicherte Daten strukturiert darzustellen und Basisfunktionalitäten eines E-Mail Clients auszuführen. Für die Indexierung der Datenbank wurde Apache Lucene eingesetzt, welches komplexe Suchanfragen unterstützt und eine hohe Performance beim Durchsuchen von riesigen Mailarchiven ermöglicht.

Durch Verwendung der CouchDB konnte eine einfache Datenstrukturierung erfolgen. CouchDB Dokumente entsprechen “in sich geschlossenen Informationen“, im Fall eines Mail-verarbeitenden System, sind dies Mail-Nachrichten. Im Prototypen mailcdb wird eine Mail-Nachricht als ein CouchDB Dokument repräsentiert. Anhänge von Mail-Nachrichten entsprechen den Anhängen eines CouchDB Dokuments. Die Simplizität dieses Datendesigns gemeinsam mit der einfach zu bedienenden REST basierenden API, sowie der gut und einfach gehaltenen Apache CouchDB Dokumentation, ermöglichen eine schnelle Entwicklung und Anbindung neuer Applikationen.

Im direkten Vergleich mit etablieren Mail-Clients wurden beim Datenimport längere Laufzeiten festgestellt. Die Inkludierung der mit Base64 kodierten Inhalte von Mail-Nachrichten in den CouchDB Dokumenten wirkt sich negativ auf die Performanz von allen Importvorgängen des Prototypen aus. Ein gleichzeitiges Hochladen von mehreren Anhängen eines Dokuments über die CouchDB Schnittstelle wird nicht unterstützt. Die

Alternative Anhänge lokal im Dateisystem zu speichern entspricht nicht dem Konzept, alle Daten in der CouchDB aufzubewahren und würde damit viele Vorteile durch die Verwendung der CouchDB zunichte machen.

So bietet der Prototyp einen hohen Grad an Interoperabilität und ermöglicht das Teilen von Informationen durch standardisierte Datenzugriffe. Integrierte Funktionalitäten von CouchDB erlauben einfache Migrationen und Replikationen und schaffen damit die Voraussetzungen für hohe Skalierbarkeit der Mail-Datenbank. Eine hohe Verfügbarkeit wird durch eine fehlertolerante Architektur von CouchDB ermöglicht. Durch Multiversion Concurrency Control werden Blockaden von Schreib- oder Lesevorgängen auch bei Verwendung von CouchDB-Clustern verhindert und Datenverlust durch die integrierte Versionierung aller Mail-Nachrichten vorgebeugt.

Eine Entwicklung des vom Prototypen vorgestellten Systems könnte weitere persönliche Daten wie Kontakte, Termine und Aufgaben, aber auch Nachrichten und Informationen aus sozialen Netzwerken in einer Oberfläche vereinen. Diese könnten in einer verteilten Anwendung zur persönlichen Informationsverwaltung genutzt werden. Ein solches System könnte als CouchApp implementiert werden, um viele der Vorteile der Datenspeicherung in CouchDB auch auf die Programmdaten der Applikation selbst anwenden zu können.

# Literatur

[1] BAZERMAN, G. mbox-0.1: Read and write standard mailbox files. http://hackage.haskell.org/packages/archive/mbox/0.1/doc/html/Data-MBox.html. Stand: Jul 2012.

[2] BERNSTEIN, D. J. Using maildir format. http://cr.yp.to/proto/maildir.html/. Stand: Aug 2012.

[3] BRYAN O'SULLIVAN, DON STEWART, J. G. *Real World Haskell*. O'Reilly Media, 2008.

[4] COUCHDB-LUCENE PROJECT. Documentation. https://github.com/rnewson/couchdb-lucene. Stand: Aug 2012.

[5] DAUGHERTY, J. Haskellnet. https://github.com/jtdaugherty/HaskellNet. Stand: Aug 2012.

[6] DOCUMENTATION, G. Maildir. http://www.gnus.org/manual/gnus_196.html#SEC196. Stand: Aug 2012.

[7] GOSPODNETIC, O. Introduction to text indexing with apache jakarta lucene. http://onjava.com/pub/a/onjava/2003/01/15/lucene.html. Stand: Aug 2012.

[8] J. CHRIS ANDERSON, JAN LEHNARDT, N. S. *CouchDB: The Definitive Guide*. O'Reilly, 2010.

[9] LEONARD RICHARDSON, S. R. *RESTful Web Services*. O'Reilly Media, 2007.

[10] LYNAGH, I. mime-string-0.4: Mime implementation for string's. http://hackage.haskell.org/package/mime-string. Stand: Jul 2012.

[11] MARLOW, S. Haddock: A haskell documentation tool. http://www.haskell.org/haddock. Stand: Aug 2012.

[12] MARTIN PORTER, R. B. German stemming algorithm. http://snowball.tartarus.org/algorithms/german/stemmer.html. Stand: Aug 2012.

[13] MICROSOFT-DEVELOPER-NETWORK. Outlook personal folders (.pst) file format. http://msdn.microsoft.com/en-us/library/ff385210.aspx. Stand: Aug 2012.

[14] NELSON, R. Mbox - file ontaining mail messages. http://qmail.org./man/man5/mbox.html. Stand: Jul 2012.

[15] NETWORK-WORKING-GROUP. The application/json media type for javascript object notation (json). http://www.ietf.org/rfc/rfc4627. Stand: Jul 2012.

[16] NETWORK-WORKING-GROUP. mbox media type. http://tools.ietf.org/html/rfc4155. Stand: Jul 2012.

[17] NETWORK-WORKING-GROUP. Mime part five: Conformance criteria and examples. http://tools.ietf.org/html/rfc2049. Stand: Jul 2012.

[18] NETWORK-WORKING-GROUP. Mime part four: Registration procedures. http://tools.ietf.org/html/rfc2048. Stand: Jul 2012.

[19] NETWORK-WORKING-GROUP. Mime part one: Format of internet message bodies. http://tools.ietf.org/html/rfc2045. Stand: Jul 2012.

[20] NETWORK-WORKING-GROUP. Mime part three: Message header extensions for non-ascii text. http://tools.ietf.org/html/rfc2047. Stand: Jul

2012.

[21] NETWORK-WORKING-GROUP. Mime part two: Media types. http://tools.ietf.org/html/rfc2046. Stand: Jul 2012.

[22] NETWORK-WORKING-GROUP. Post office protocol - version 3. http://tools.ietf.org/html/rfc1939. Stand: Jul 2012.

[23] PROJECT, G. Glade - a user interface designer. http://glade.gnome.org/. Stand: Aug 2012.

[24] SCHELIGA, M. *CouchDB kurz und gut*. O'Reilly, 2010.

[25] STEPHEN BLACKHEATH, MATTHEW ELDER, J. S. Smtpclient: A simple smtp client library. http://hackage.haskell.org/package/SMTPClient. Stand: Aug 2012.

[26] TEAM, G. A gui library for haskell based on gtk. http://www.haskell.org/haskellwiki/Gtk2Hs. Stand: Aug 2012.

[27] WIKI, C. Http document api. http://wiki.apache.org/couchdb/HTTP_Document_API. Stand: Aug 2012.

[28] WIKI, C. Technical overview. http://wiki.apache.org/couchdb/Technical%20Overview. Stand: Aug 2012.

[29] WIKI, H. Libraries and tools/mime strike force. http://www.haskell.org/haskellwiki/Libraries_and_tools/MIMEStrikeForce. Stand: Aug 2012.

[30] ZHOU, D. P. Delve inside the lucene indexing mechanism. http://www.ibm.com/developerworks/library/wa-lucene/. Stand: Sept 2012.

# Abbildungsverzeichnis



# Tabellenverzeichnis



# Listings

# Anhang

## A. Installation

### A.1. Software-Pakete

Für die Installation des Prototypen wird das mailcdb Package und folgende Software-Pakete benötigt. Nachfolgende Befehle installieren alle Abhängigkeiten für die Installation unter Linux Ubuntu 12.10.

```
apt-get update
sudo apt-get install ghc7 cabal-install gtk2hs-buildtools glade
   -gnome

sudo apt-get install erlang-dev erlang-eunit erlang-nox
sudo apt-get install libmozjs185-dev libicu-dev libcurl4-gnutls
   -dev libtool
```

**Listing 9:** Installation Software-Pakete

### A.2. Apache CouchDB

Der Quellcode von Apache CouchDB der aktuellen Version 1.2 kann über die Website bezogen werden. Folgend die Befehle zur Kompilierung des Programms:

```
wget http://tweedo.com/mirror/apache/couchdb/releases/1.2.0/
   apache-couchdb-1.2.0.tar.gz
tar xfvz apache-couchdb-1.2.0.tar.gz
cd apache-couchdb-1.2.0/
./configure
```

```
make && sudo make install
```

**Listing 10:** Installation CouchDB

Will man CouchDB als Dienst automatisch starten lassen sind folgende Befehle notwendig:

```
sudo adduser --disabled-login --disabled-password --no-create-
    home couchdb

sudo chown -R couchdb:couchdb /usr/local/var/log/couchdb
sudo chown -R couchdb:couchdb /usr/local/var/lib/couchdb
sudo chown -R couchdb:couchdb /usr/local/var/run/couchdb

sudo ln -s /usr/local/etc/init.d/couchdb  /etc/init.d
sudo update-rc.d couchdb defaults
```

**Listing 11:** CouchDB als Dienst konfigurieren

CouchDB läuft standardmäßig auf Port 5984 und kann über `http://localhost:5984` erreicht werden.

## A.3. CouchDB-Lucene

Um das Apache Lucene Framework zu nutzen muss die JAVA-Bibliothek CouchDB-Lucene installiert werden. Die aktuelle Version findet man im GitHub und kann wie folgt installiert und gestartet werden.

```
sudo apt-get install git-core maven2 openjdk-6-jdk
cd <target-directory>
git clone git://github.com/rnewson/couchdb-lucene.git
cd couchdb-lucene
mvn
```

```
cd target
sudo unzip couchdb-lucene-<version>-dist.zip
cd couchdb-lucene-<version>
sudo ./bin/run
```

**Listing 12:** Installation CouchDB-Lucene

Damit CouchDB mit couchdb-lucene kommunizieren kann müssen folgende Zeilen in der Datei */usr/local/etc/couchdb/local.ini* ergänzt werden:

```
[external]
fti=<target-directory>/couchdb-lucene/couchdb-external-hook.py

[httpd_db_handlers]
_fti = {couch_httpd_external, handle_external_req, <<"fti">>}
```

**Listing 13:** Konfiguration CouchDB für Lucene

CouchDB-lucene läuft standardmäßig auf Port 5985 und kann über `http://localhost:5985` erreicht werden.

## A.4. Modul: HaskellNet

Das zum Zeitpunkt der Entwicklung von Mailcdb verfügbare haskellNet Package ist die Version 0.3. Diese Version weist Kompilierungs-Probleme für GHC ab Version 7.0 auf und lässt sich nicht installieren. Im GitHub unter https://github.com/jtdaugherty/HaskellNet kann eine bereinigte Version bezogen werden bei der Abhängigkeiten zu den obsoleten haskell98 Modul entfernt wurden (`https://nodeload.github.com/jtdaugherty/HaskellNet/legacy.zip/master`). Nach dem Entpacken des Tarballs kann das HaskellNet Modul mittels Cabal installiert werden.

```
tar xfvz jtdaugherty-HaskellNet-3a029c8.tar.gz
```

```
cd jtdaugherty-HaskellNet-3a029c8
cabal update
cabal install
```

**Listing 14:** Installation HaskellNet

## A.5. Modul: Mailcdb

Das mailcdb Package und dessen Abhängigkeiten werden mittels Cabal installiert. Vor dem Start der ausführbare Datei *mailcdb* muss sichergestellt werden, dass das Verzeichnis *resources* und dessen Inhalt im aufrufenden Verzeichnis existiert und die benötigten Design Dokumente in CouchDB erstellt wurden, der Aufruf mit den Parametern "db prepare" legt diese Design Dokumente automatisch an.

```
tar xfvz mailcdb-0.9.tar.gz
cd mailcdb-0.9
cabal install

cp dist/build/mailcdb/mailcdb ./
./mailcdb db prepare
./mailcdb
```

**Listing 15:** Installation Mailcdb

Für den Versand von E-Mails per SMTP ist die Angabe eines SMTP Servers und einer Absenderadresse in der Konfigurations-Datei *resources/mailcdb.cfg* notwendig. Bei folgender Konfiguration wurde ein lokal installierter SMTP Server (z.B: Postfix) verwendet.

```
# Options for the SMTP Server
[SMTP]
hostname = 127.0.0.1
sender = joe@example.com
```

**Listing 16:** mailcdb Konfigurationsdatei

Folgend das Listing von mailcdb.cabal:

```
name: mailcdb
version: 1.0
cabal-version: >= 0.2
build-type: Simple
license: BSD3
license-file: LICENSE
copyright: Karl Beranek 2011
maintainer: Karl Beranek
build-depends: HaskellNet -any, HTTP >=4000.0.4, base -any,
   json -any, utf8-string -any, mime-string -any, network -any,
    time -any, bytestring -any, gtk -any, glade -any, mtl -any,
    ConfigFile -any, text -any, old-time -any, old-locale -any,
    base64-string -any

stability: experimental
synopsis: The couchDB-Mail Package
description:  the couchdb based client implemented in haskell
   for search and retrieval of e-mail
category: Testing
author: Karl Beranek
data-files: gui.glade mailcdb.cfg readme.txt
data-dir: "resources"
exposed-modules: Mailcdb Main Mail.POP3 Mail.SMTP Mail.MBOX
   Types.Account Types.Mail Database.CouchDB Database.CouchDB.
   Unsafe Database.CouchDB.HTTP Database.CouchDB.JSON UI.Helper
    UI.CLI UI.Types UI.GUI DB.CDBHelper DB.CDBHandler DB.DAL
exposed: False
buildable: True
```

```
hs-source-dirs: src

executable: mailcdb
main-is: Main.hs
buildable: True
hs-source-dirs: src
```

**Listing 17:** mailcdb.cabal

# B. Handbuch

## B.1. Command Line Interface

Der Prototyp mailcdb bietet folgende Kommandozeilenoptionen.

```
./mailcdb help
System for storing mail-messages in the restfull document
   oriented database couchdb.

Usage: mailcdb gui
       mailcdb db prepare
       mailcdb account new
       mailcdb account list
       mailcdb account enable <accountLabel>
       mailcdb account disable <accountLabel>
       mailcdb mail new <to> <subject> <body>
       mailcdb mail fetch
       mbox <filePath>
       mailcdb help
```

**Listing 18:** Kommandozeilenoptionen

Das Hinzufügen eines POP3 Kontos gestaltet sich wie folgt:

```
./mailcdb account new
Please enter details about your POP3 mail account!

Host: example.com
Username: joe
Password: secret
E-Mail: joe@example.com
Label: example.com
```

```
Mail account stored!
```

**Listing 19:** Hinzufügen eines POP3 Kontos

## B.2. Graphische Benutzeroberfläche

Startet man den Prototypen ohne Parameter wird die grafische Benutzeroberfläche geladen. Die Oberfläche teilt sich wie folgt auf: Kontextabhängige Aktionstasten befinden sich am oberen Rand des Fensters, eine Übersicht der vorhandenen Labels (Label-Listenfeld) und damit verknüpften E-Mails (Mail-Listenfeld) sowie ein Such-Eingabefeld sind auf der linken Seite angeordnet. Die rechte Seite zeigt wesentliche Informationen der ausgewählten E-Mails wie Absender, Empfänger, Betreff, Anhänge und Mailparts und ein Vorschaufenster in dem Plaintext-Mailparts angezeigt werden (siehe Abbildung 5).

Neu erstellte E-Mails erhalten das Label *Drafts* und können im Vorschaufenster bearbeitet werden. Es können Plaintext E-Mails mit Klartext Inhalt gespeichert und versendet werden. Gesendete E-Mails erhalten das Label *sent*, E-Mails die temporär nicht versendet werden konnten erhalten das Label *outbox*.

In *mailcdb* können Mailnachrichten von MBOX-Dateien oder von POP3 Servern importiert werden. Letztere Quelle erfordert die Angabe verschiedener Parameter des POP3 Servers mittels CLI Aufruf (siehe Listing 19). Importierte Mailnachrichten erhalten als Label den Namen des Mailkontos bzw. den Dateinamen der MBOX-Datei. Der Import von neuen E-Mails erfolgt als separat gestarteter Prozess und erfolgt im Hintergrund. Der Verlauf und das Ende des Imports werden durch Meldungen in der Statusleiste angezeigt. Beim nächsten Wechsel zu einer weiteren Nachricht oder nach manueller Aktualisierung werden die neuen Nachrichten angezeigt.

Im Eingabefeld für Suchanfragen kann eine Volltextsuche mittels Apache Lucene gestartet werden die E-Mails und Anhänge durchsucht. Die Suchresultate werden im

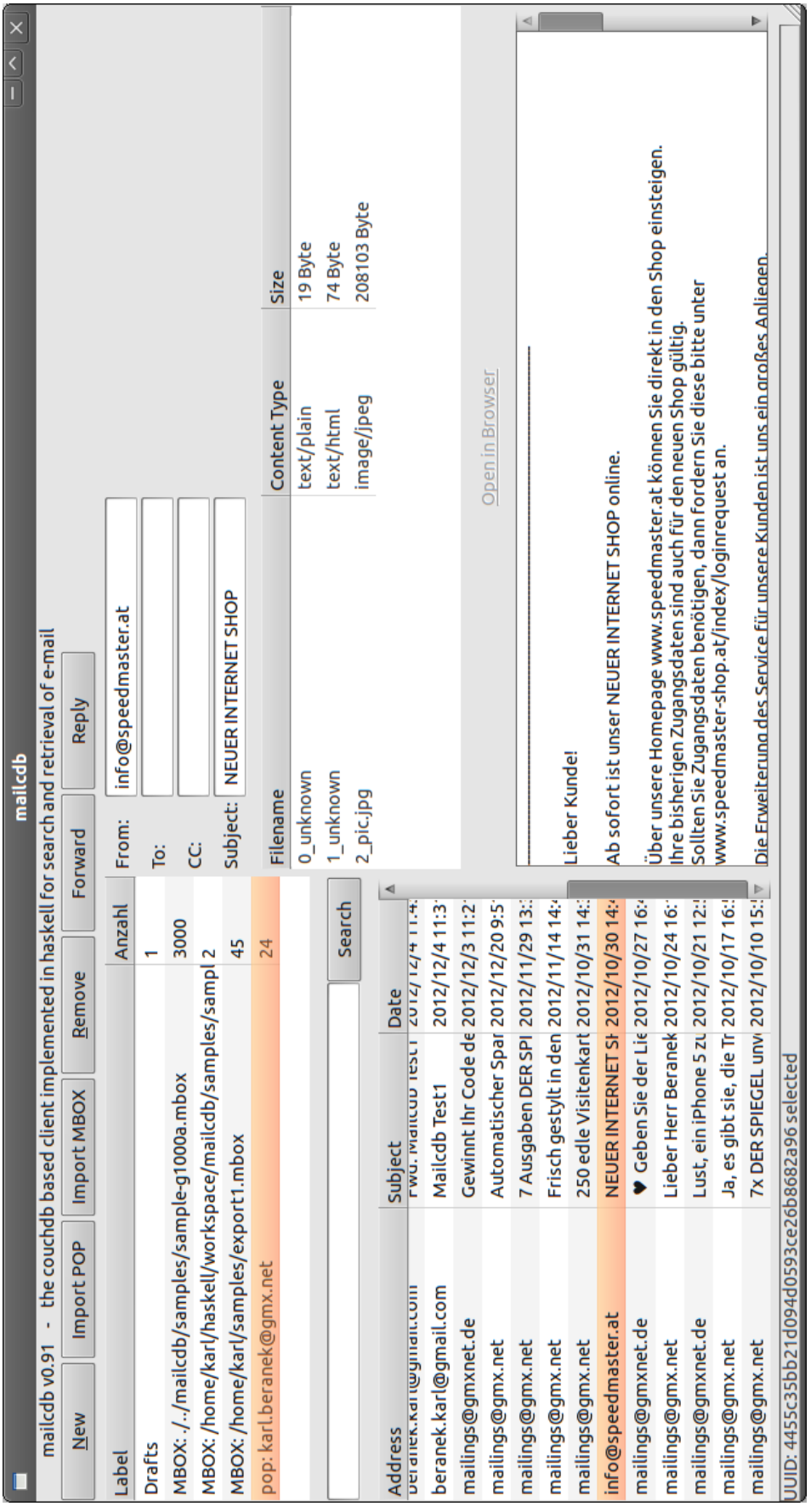


**Abbildung 5:** Screenshot des Prototypen mailcdb

Listenfeld darunter angezeigt. Apache Lucene bietet diverse Suchparameter für komplexere Suchanfragen, so kann mit der Eingabe “subject:<Betreff>” nur nach E-Mails mit einem bestimmten Betreff gesucht werden. Weitere benannte Attribute nach denen direkt im Index gesucht werden kann sind “to” oder “from”.

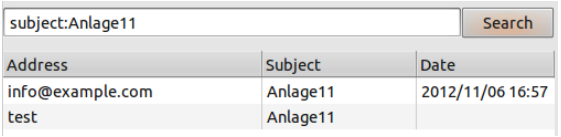


**Abbildung 6:** Screenshot der Suchmaske

Sobald eine Nachricht im Mail-Listenfeld ausgewählt wird, werden Details wie Absenderadresse, Empfängeradresse(n) und Betreff in den jeweiligen Feldern angezeigt. Relevante Informationen von Mailanhängen der Nachricht können den Listenfeld für Anhänge entnommen werden. Für jeden Anhang wird die URL angezeigt unter der der Kontent in einem Browserfenster betrachtet werden kann.

# C. Dokumentation des Quellcode

# mailcdb-1.0: The couchDB-Mail Package

the couchdb based client implemented in haskell for search and retrieval of e-mail

## Modules

# DB.CDBHandler

Safe Haskell None

This modul handels connections to Apache CouchDB at a high level. It provides functions to conveniently manipulate Documents, views and indexViews created for Apache Lucene in a couchDB.

Contents



## Database functions

```
createMyDB :: IO ()
```

## Document functions

**updateDocument**

```
:: Data a
=> DB                              database name
-> IDString                        document UUID
-> a                               new document
-> IO (Either IOError Bool)
```

Update couchdb document

return Right True on success; return Left errorString on error

**deleteDocument**

```
:: DB       database name
-> Doc      document id
-> IO ()
```

Delete couchdb document identified by document id

**storeDocument**

```
:: Data a
=> DB              database name
-> a               document
-> IO IDString     UUID of created document
```

Write couchdb document, return written UUID

**retrieveDocument**

```
:: Data a
=> DB                              database name
-> IDString                        document UUID
-> IO (Either ErrorString a)       document
```

Retrieve couchdb document by given id

return Right Doc on success; return Left errorString on error

---

**retrieveDocFromView**

```
:: Data a
=> DB                                database name
-> Doc                               document id
-> ViewString                        index view
-> QueryPairs                        query parameters
-> IO (Either ErrorString [a])
```

Retrieve couchdb document from db on view

return Right Doc on success; return Left errorString on error

---

**retrieveIdDocFromView**

```
:: Data a
=> DB                                database name
-> Doc                               document id
-> ViewString                        index view
-> QueryPairs                        query parameters
-> IO (Either ErrorString [(Doc, a)])
```

Retrieve (id,doc) from db on view

return Right Doc on success; return Left errorString on error

---

**storeJSObject**

```
:: DB            database name
-> JSValue       JSObject
-> IO IDString   UUID of created document
```

Write JSObject to couchdb, return written UUID

## Query functions

---

**queryDB**

```
:: DB                      database name
-> ViewString              index view
-> [(String, String)]      search parameters
-> IO [QueryViewResult]    search results array
```

query cdb with given search parameters, return search results array

---

**queryDBKeys**

| | |
|---|---|
| `:: DB` | database name |
| `-> ViewString` | index view |
| `-> [(String, String)]` | search parameters |
| `-> IO [IDString]` | search results array |

query cdb with given search parameters, return keys only

## Helper functions

**`cleansJSValue :: JSValue -> JSValue`**

removes id from JSValue (new id is provided by db)

## Type synonyms

`type` **`IDString`** `= String`

`type` **`QueryViewResult`** `= (Doc, JSValue)`

`type` **`ErrorString`** `= String`

## Predefined strings

**`designdocAccount :: Doc`**

**`designdocMail :: Doc`**

**`mailDB :: DB`**

**`mailDBString :: [Char]`**

# DB.CDBHelper

Safe Haskell None

This modul creates design docs in couchDB. Javascript functions for map, reduce, fti, and validation functions are defined as ViewMap, ViewMapReduce, FTIFunction and ValidFunction constuctors. These elements are encapsulated in a CouchDB computation and run by runCouchDB'

## Documentation

**setMailViews**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create views with defined map functions for mails

**setAccountViews**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create views with defined map functions for accounts

**setFTI**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create design doc for index functions

**setValidation**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create design doc for validation function

# DB.DAL

Safe Haskell None

Database Abstraction Layer: provides type safe functions for accessing couchdb

Contents



## Document functions

**prepareDB** :: IO ()

Create database, design documents for mail, accounts, full text search, and validation and store sample mail account

**storeMail**

:: Mail — mail
-> IO IDString — UUID of created document

Store mail to cdb

**storeMailInc**

:: Mail — mail
-> IO ()

Store mail to cdb and increment countRetr in accountDB

**storeAccount**

:: Account — account to store
-> IO IDString — UUID of created document

Store pop3Account to cdb

**retrieveHeader**

:: IDString — UUID of part to retrieve
-> IO (Either ErrorString Header) — errorstring or retrieved part

Retrieve mailpart by given id

return Right Part on success; return Left ErrorString on error

**retrieveAttachments**

:: IDString — UUID of attachment to retrieve
-> IO (Either ErrorString [Attachment]) — errorstring or retrieved attachment

Retrieve mail-attachments by given id

return Right list of attachments on success; return Left ErrorString on error

**updateHeader**

```
:: IDString                        UUID of mailpart to update
-> Header                          updated instance of mailpart to store
-> IO (Either IOError Bool)        errorstring or True
```

Update mailpart

return Right True on success; return Left ErrorString on error

---

**deleteMail**

```
:: IDString    UUID of mailpart to delete
-> IO ()
```

Delete mailpart with given UUID

---

**enableMailAccount**

```
:: Label    label of mail account to enable
-> IO ()
```

enable mail account

---

**disableMailAccount**

```
:: Label    label of mail account to disable
-> IO ()
```

disable mail account

---

**mailAccountStatus**

```
:: Label    label name
-> Bool     activation status
-> IO ()
```

set activation status of mail accounts in cdb (enable or disable)

## Query view functions

---

**retrieveEnabledMailAccounts** `:: IO (Either String [Account])`

Retrieve enabled mail accounts (p_enabled=true)

return Right [Account] on success; return Left ErrorString on error

---

**findAccountsByLabel**

```
:: Label                                          label name
-> IO (Either ErrorString [(Doc, Account)])
```

Retrieve account with given label name

return Right [ID, Account] on success; return Left ErrorString on error

---

**retrieveMailInfoRec**

```
:: Label                                    label name
-> IO (Either ErrorString [MailInfo])       errorString or list of mailInfo
```

Retrieve mailInfo of received mails with given label name (field mailaddress = m_From)

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveMailInfoSent**

```
:: Label                                    label name
-> IO (Either ErrorString [MailInfo])       errorString or list of mailInfo
```

Retrieve mailInfo of outgoing or sent mails with given label name (field mailaddress = m_To )

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveMailInfo**

```
:: IDString                                 UUID of mailInfo to retrieve
-> IO (Either ErrorString [MailInfo])       errorString or list of mailinfo
```

Retrieve mailInfo by given id

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveLabels** `:: IO [(String, Int)]`

Retrieve name and number of existing labels from cdb

---

**retrieveAccounts**

```
:: IO (Either ErrorString [Account])    errorString or list of accounts
```

Retrieve all accounts from cdb

return Right [Account] on success; return Left ErrorString on error

## Query full-text-search functions

---

**queryFulltext**

```
:: String                   querytext
-> IO [QueryViewResult]     result of query queryFulltext queryText = queryDB mailDB header [(q, queryText)] carefull!!!
```

fulltext-query DB for queryText

---

**queryFulltextKeys**

```
:: String           querytext
```

-> IO [IDString]  key only result queryFulltextKeys queryText = queryDBKeys mailDB header [(q, queryText)] carefull!!!

fulltext-query DB for queryText, return keys only

---

```
type Label = String
```

## Helper functions

---

```
incrementCountRetr :: Label -> IO ()
```

Increment number of retrieved mails (countRetr) in cdb

---

```
addAttachment2JSON :: JSValue -> [Attachment] -> JSValue
```

Add attachment to JSON Object

---

```
type IDString = String
```

---

```
type ErrorString = String
```

---

```
maildbURL :: [Char]
```

---

# Mail.MBOX

Safe Haskell None

Mbox handling functions

Example of opening a mbox file:

```
module Main where
import ...

main :: IO ()
main = do
    let filePath = sample1.mbox
    mails <- doImportMbox filePath
    sample code
    sample code
    sample code
```

## Documentation

**doImportMbox**

```
:: FilePath                      file path
-> IO (Either String [Mail])     errorString or list of mails
```

**splitMailString**

```
:: String      MBOX string
-> [String]    list of mail strings
```

Parse MBOX string, separate different mail strings

# Mail.POP3

Safe Haskell None

This module handels connections to POP3-Server

Example of downloading messages from a pop3 server:

```
module Main where
import ...

mailAccount = POP3Account {
                    p_host      = pop.example.com,
                    p_user      = user@example.com,
                    p_pass      = password,
                    p_email     = user@example.com,
                    p_label     = label,
                    p_countRetr = 100,
                    p_enabled    = True }

main :: IO ()
main = do
    mails <- retrieveMailFromAccount mailAccount
    sample code
    sample code
    sample code
```

Contents



## Datatypes

data **POP3Handler**

Datatype encapsulating all IO functions for pop3 handling

**Constructors**

**POP3Handler**

**ph_conn :: IO ()**
open connection to pop3 server

**ph_disconn :: IO ()**
close connection to pop3 server

**ph_stat :: IO Int**
return total number and size of emails on server

**ph_list :: IO ()**
print size of 1st email

**ph_new :: IO Int**
return number of new emails on server

**ph_fetch :: Int -> IO [ByteString]**
fetch emails starting at given number to total number of emails on server

**connectToPopServerWith**

:: Account — the POP3 account to be queried

-> IO POP3Handler  return POP3Handler

Constructor of `POP3Handler` holding implementations of IO

## Mail Handling Functions

**retrieveMailFromAccount**

| | |
|---|---|
| :: Account | pop3Account |
| -> IO (Either String [Mail]) | return Right [[Mail]] on successful retrieval, Left ErrorString on error |

Convert mime-messages returned from `getNewMessages` to Types.Mail, add label (for categorizing) to mails.

return Right constructor with list of mails on success; return Left constructor with errorString on error

**getNewMessages**

| | |
|---|---|
| :: Account | pop3Account holding pop3 login info and # of retrieved mails |
| -> IO [Message] | new messages as MIME-Message |

Initiate pop3handler with given pop3account, connect, fetch mail byte stream of new mails (according to pop3Account), parse it and return as MIME-message.

Errors trigger IO exception.

# Mail.SMTP

Safe Haskell None

Sending mailparts via SMTP

## Documentation

**doSendMimeMail**

```
:: Header                          mail-header
-> String                         path to attachment
-> String                         hostname of smtp server
-> String                         mailaddress of sender
-> IO (Either IOError Bool)
```

Send mime-mail with given mailpart using SMTP

return Right True on success; return Left constructor with errorString on error

# Mailcdb

Safe Haskell None

Top-level code

Implements high level functions to manipulate mails stored in db and to import new mails to db.

Contents



## Mail manipulating functions

**runNewMail**

```
:: IO IDString
```

IDString of created mailpart

Create new empty mailpart and store in db

return IDString of created mailpart

**runNewMailReply**

```
:: IDString
-> IO (Either String (Header, IDString))
```

IDString of mailpart to cite in new mailpart

Create mail-reply by citing existing mailpart and store in db

return Left ErrorString on error; return Right mailpart and IDString of newly created mailpart as tuple

**runNewMailForward**

```
:: IDString
-> IO (Either String (Header, IDString))
```

IDString of mailpart to cite in new mailpart

Create mail-forward by citing existing mailpart and store in db

return Left ErrorString on error; return Right mailpart and IDString of newly created mailpart as tuple

**runSendMail**

```
:: IDString
-> IO (Either ErrorString LabelString)
```

IDString of mailpart to send

Errorstring or name of assigned label

Send body of mailpart specified by IDString as mail

return Left ErrorString when mail can not be retrieved from db, when insufficient mail entries are set, or when an invalid server hostname is given in configfile; return Right outbox-labelname when mail cannot send due to smtp or connection problems; return Right sent-labelname when mail was successfully transmitted to smtp server

**runDeleteMail**

```
:: IDString
-> IO ()
```

IDString of parent mailpart to delete

Remove parent mailpart and associated children-mailparts from db

## Mail importing functions

**runImportMboxFrom**

```
:: FilePath    filepath to mbox file
-> IO ()
```

Open mbox-file with given filepath, convert containing Message-Strings to Types.Mail.Part, store to db

**runRetrieveMailFromActivePopAccounts** `:: IO ()`

Retrieve all enabled POP3 accounts from db, query POP3 server, retrieve mails, and store them to db

## Helper functions

**labelNameSent** `:: [Char]`

label-name for sent mails

**labelNameDrafts** `:: [Char]`

label-name for draft-mails

**labelNameOutbox** `:: [Char]`

label-name for mails to be sent

**runConditionFunction** `:: Show t => IO (Either t t1) -> IO a -> (t1 -> IO ()) -> IO ()`

run functions depending on return of conditionFunction: success (function runOnRight) or fail (function runOnLeft)

**runConditionFunction'** `:: Show t => IO (Either t [t1]) -> IO a -> (t1 -> IO ()) -> IO () -> IO ()`

run functions depending on return of conditionFunction: success (function runOnRight) or fail (function runOnLeft) or emptyList

## DB.DAL call-through functions

### Document functions

**prepareDB** `:: IO ()`

Create database, design documents for mail, accounts, full text search, and validation and store sample mail account

**storeMail**

```
:: Mail          mail
```

```
-> IO IDString
```
UUID of created document

Store mail to cdb

---

**storeMailInc**

```
:: Mail
-> IO ()
```
mail

Store mail to cdb and increment countRetr in accountDB

---

**storeAccount**

```
:: Account
-> IO IDString
```
account to store

UUID of created document

Store pop3Account to cdb

---

**retrieveHeader**

```
:: IDString
-> IO (Either ErrorString Header)
```
UUID of part to retrieve

errorstring or retrieved part

Retrieve mailpart by given id

return Right Part on success; return Left ErrorString on error

---

**retrieveAttachments**

```
:: IDString
-> IO (Either ErrorString [Attachment])
```
UUID of attachment to retrieve

errorstring or retrieved attachment

Retrieve mail-attachments by given id

return Right list of attachments on success; return Left ErrorString on error

---

**updateHeader**

```
:: IDString
-> Header
-> IO (Either IOError Bool)
```
UUID of mailpart to update

updated instance of mailpart to store

errorstring or True

Update mailpart

return Right True on success; return Left ErrorString on error

---

**deleteMail**

```
:: IDString
-> IO ()
```
UUID of mailpart to delete

Delete mailpart with given UUID

---

**enableMailAccount**

```
:: Label    label of mail account to enable
-> IO ()
```

enable mail account

---

**disableMailAccount**

```
:: Label    label of mail account to disable
-> IO ()
```

disable mail account

---

**mailAccountStatus**

```
:: Label    label name
-> Bool     activation status
-> IO ()
```

set activation status of mail accounts in cdb (enable or disable)

## Query view functions

---

**retrieveEnabledMailAccounts** `:: IO (Either String [Account])`

Retrieve enabled mail accounts (p_enabled=true)

return Right [Account] on success; return Left ErrorString on error

---

**findAccountsByLabel**

```
:: Label                                    label name
-> IO (Either ErrorString [(Doc, Account)])
```

Retrieve account with given label name

return Right [ID, Account] on success; return Left ErrorString on error

---

**retrieveMailInfoRec**

```
:: Label                                label name
-> IO (Either ErrorString [MailInfo])   errorString or list of mailInfo
```

Retrieve mailInfo of received mails with given label name (field mailaddress = m_From)

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveMailInfoSent**

```
:: Label                                label name
-> IO (Either ErrorString [MailInfo])   errorString or list of mailInfo
```

Retrieve mailInfo of outgoing or sent mails with given label name (field mailaddress = m_To )

return Right [MailInfo] on success; return Left ErrorString on error

**retrieveMailInfo**

```
:: IDString                                  UUID of mailInfo to retrieve
-> IO (Either ErrorString [MailInfo])        errorString or list of mailinfo
```

Retrieve mailInfo by given id

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveLabels** `:: IO [(String, Int)]`

Retrieve name and number of existing labels from cdb

---

**retrieveAccounts**

```
:: IO (Either ErrorString [Account])   errorString or list of accounts
```

Retrieve all accounts from cdb

return Right [Account] on success; return Left ErrorString on error

## Query full-text-search functions

---

**queryFulltext**

```
:: String                  querytext
-> IO [QueryViewResult]    result of query queryFulltext queryText = queryDB mailDB header [(q, queryText)] carefull!!!
```

fulltext-query DB for queryText

---

**queryFulltextKeys**

```
:: String          querytext
-> IO [IDString]   key only result queryFulltextKeys queryText = queryDBKeys mailDB header [(q, queryText)] carefull!!!
```

fulltext-query DB for queryText, return keys only

---

`type` **Label** `= String`

## Helper functions

---

**incrementCountRetr** `:: Label -> IO ()`

Increment number of retrieved mails (countRetr) in cdb

---

**addAttachment2JSON** `:: JSValue -> [Attachment] -> JSValue`

Add attachment to JSON Object

---

`type` **IDString** `= String`

```
type ErrorString = String
```

```
maildbURL :: [Char]
```

# Main

Safe Haskell None

Program entry point

## Documentation

```
main :: IO ()
```

Run mailcdb with given list of arguments. When no arguments are given mailcdb starts with gtk-gui.

```
Usage: mailcdb gui                                  Start mailcdb with g
       mailcdb db prepare                           Create new db in cou
                                                    design documents for
                                                    text search, and val
       mailcdb account new                          Create new pop3 mail
       mailcdb account list                         List pop3 mail accou
       mailcdb account enable <accountLabel>        Enable pop3 mail acc
       mailcdb account disable <accountLabel>       Disable pop3 mail ac
       mailcdb mail new                             Create new Mail
       mailcdb mail fetch                           Fetch mails from ena
       mbox <filePath>                              Import mails from gi
       mailcdb help                                 Display usage info
```

# Types.Account

Safe Haskell Safe-Inferred

Types that are commonly used in mailcdb

## Documentation

```
data Account
```

Record which contains all information to connect and authenticate to a POP3 server and stores the number of retrieved emails

### Constructors

**POP3Account**

**p_host :: String**
IP or hostname of pop3 server

**p_user :: String**
username of valid email account

**p_pass :: String**
password of valid email account

**p_email :: String**
full email address

**p_label :: String**
label to be used for storing emails of this account

**p_countRetr :: Int**
number of mails retrieved so far

**p_enabled :: Bool**
switch for enabling or disabling this account

### Instances

Eq Account
Data Account
Ord Account
Read Account
Show Account
Typeable Account

```
samplePOP3Account :: Account
```

sample mail account

# Types.Mail

Safe Haskell None

Types that are commonly used in mailcdb

## Documentation

```
type LabelString = String
```

```
data Mail
```

A mail consists of a header and a list of attachments

### Constructors

**Mail**

**m_header :: Header**
Mailheader

**m_parts :: [Attachment]**
List of attachments

### Instances

Show Mail

```
data Header
```

Mail header represented as algebraic data type, holding header information

### Constructors

**Header**

**h_from :: String**
From header

**h_body :: String**
body

**h_to :: String**
recipient

**h_cc :: String**
cc field

**h_label :: [String]**
list of labels

**h_sender :: String**
sender

**h_date :: String**
sender date

**h_mime :: String**

mime version

**h_subject :: String**
subject of mail

**h_contentType :: String**
content type

**h_contentTypeParameter :: String**
content type parameter

**_id :: String**
uuid in couchdb

### Instances

Eq Header
Data Header
Ord Header
Read Header
Show Header
Typeable Header

---

data **Attachment**

Type representing attachments of mails

### Constructors

**Attachment**

**a_filename :: String**
filename of attachment

**a_type :: String**
type of attachment

**a_content :: String**
body

**a_length :: Int**
length of body

### Instances

Eq Attachment
Data Attachment
Ord Attachment
Read Attachment
Show Attachment
Typeable Attachment

---

data **MailInfo**

This type provides minimal info about mail

### Constructors

**MailInfo**

**mId :: String**
uuid in couchdb

**mAddress :: String**
mail address (to or from)

**mSubj :: String**
subject

**mDate :: String**
sender date

**mContentType :: String**
content type

### Instances

Data MailInfo

Show MailInfo

Typeable MailInfo

---

**emptyHeader** :: Header

Construct empty header type

---

**writeLabel** :: String -> Mail -> Mail

Add given string as label to mail

---

**convertMessageToMail**

:: Message — mime-mail message (Codec.MIME.String.Message) to be converted
-> Mail — returned mail message of type Types.Mail

Convert mail message of type Codec.MIME.String.Message to type Types.Mail

---

**incrementFilenames**

:: Mail
-> Mail

make filenames of attachments unique

---

**parseInfo**

:: MessageInfo
-> Header — mail message with parsed info
-> Header — returned mail message of type Types.Header

Parse MessageInfo of mime-mail and write to type Types.Header

**parseContent**

```
:: MessageContent
-> Header
-> Mail                  returned list of mail messages
```

Parse MessageContent of mime-mail and write to mail message of type Types.Mail

when content is some sort of multipartcontent function `convertMessageToMail` is called recursivly and multipart-mails are attached to returned list of mails

**parseAttachment**

```
:: [Message]
-> [Attachment]    returned list of attachments
```

Parse parts of mime-mail and return list of type Types.Attachment

**month** `:: [[Char]]`

Convert textual presentation of month to numeric

**month2Numeric** `:: Month -> String`

# UI.CLI

Safe Haskell None

This module provides a command line interface for testing some functions

## Documentation

**readmeFile** :: [Char]

**runCLI** :: IO ()

Main function for using the command line interface

**cliListAccounts** :: IO ()

List all accounts in db

**cliNewAccount** :: IO ()

Store new account to db

**cliNewMail** :: String -> String -> String -> IO ()

Create new mail and store to db

**enterMailInfo** :: IO [String]

Collect mail information from inputs

**main_example** :: IO ()

Store new account, enable it, retrieve messages, and store them do db

# UI.GUI

Safe Haskell None

This module provides a GTK user interface

## Documentation

**runGUI** :: IO ()

Main GTK+ loop

**loadGlade**

:: FilePath filepath of glade file
-> IO GUI

Load and define all gtk elements

**connectGui** :: GUI -> IO (ConnectId Button)

Setup all event handler for GTK+ elements

# UI.Helper

Safe Haskell None

action handler for buttons

Contents



## Widget action handler

**buttonReply**

```
:: GUI
-> ListStore LabelInfo
-> ListStore MailInfo
-> IDString
-> IO ()
```

Action handler for button `reply`: create new mail-reply and select it

**buttonImportPop**

```
:: GUI
-> IO ()
```

Action handler for button 'import pop': start importing mails of all stored mail accounts, when finished change button label to `refresh`

**buttonForward**

```
:: GUI
-> ListStore LabelInfo
-> ListStore MailInfo
-> IDString
-> IO ()
```

Action handler for button `forward`: create new mail-reply and select it

**buttonSend**

```
:: GUI
-> ListStore MailInfo
-> ListStore LabelInfo
-> IO ()
```

Action handler for button `send`: start sending an email

**buttonRemove**

```
:: GUI
-> ListStore MailInfo
-> ListStore LabelInfo
-> ListStore a

-> IO ()
```

Action handler for button `remove`: delete selected mail from db

---

**buttonNew1**

```
:: GUI
-> ListStore LabelInfo
-> ListStore MailInfo
-> IO ()
```

Action handler for button `new`: store new mail to db and select it

---

**buttonSave**

```
:: GUI
-> ListStore MailInfo
-> ListStore LabelInfo
-> IO ()
```

Action handler for button `save`: save data from mail input widgets to db

---

**buttonImportMbox**

```
:: GUI
-> FilePath
-> IO ()
```

Action handler for button 'import mbox': start importing mails from mbox, when finished change button label to `refresh`

---

**buttonSearch**

```
:: GUI
-> ListStore MailInfo
-> IO [()]
```

Action handler for button `search`: get text from text entry, start db query, return results to treeview `mails`

---

**openOpenFileChooser**

```
:: Window
-> IO (Maybe FilePath)
```

Open file chooser dialog for file opening (select mbox)

## TreeView action handler

---

**treeviewAttachmentsSelectionChanged**

```
:: ListStore Attachment
-> t
-> GUI
-> IO ()
```

Handler for event `selection_changed` of treeview `attachments`

---

**treeviewLabelsSelectionChanged**

```
:: GUI
-> ListStore LabelInfo
-> t
-> ListStore MailInfo
-> ListStore a
-> IO ()
```

Handler for event `selection_changed` of treeview `labels`: adjust button sensitivities, set appState according to selected label, update treeview `mails`

---

**treeViewMailSelectionChanged**

```
:: ListStore MailInfo
-> GUI
-> ListStore Attachment
-> t
-> IO ()
```

Handler for event `selection_changed` of treeview `mails`: adjust button sensitivities, update treeview `attachments`

---

**populateTreeviewLabels**

```
:: ListStore LabelInfo
-> IO ()
```

Populate info to treeview `labels`

## Set widget attributes

---

**setTreeviewAtributes**

```
:: (TreeViewClass self, TreeModelClass (model row), TypedTreeModelClass model)
=> self
-> model row
-> String
-> Int
-> (row -> [AttrOp CellRendererText])
-> IO ()
```

Set several common attributes for treeview widgets

---

**setMailDetailsEditable**

```
:: GUI
-> Bool
-> IO ()
```

Set widget's editable flag (r or rw-mode)

---

**pushStatusText**

```
:: GUI
-> String
-> IO ()
```

Push given text to statusbar

---

**emptyMailDetail**

```
:: GUI
-> IO ()
```

Empty mail details widgets

## Helper functions

---

**retrieveLabels** `:: IO [(String, Int)]`

Retrieve name and number of existing labels from cdb

---

**convertToLabelInfo**

```
:: [(String, Int)]
-> [LabelInfo]
```

Convert list of pairs to type LabelInfo

---

# UI.Types

Safe Haskell None

Types that are commonly used in UI.* Modules

## Documentation

**gladeFile** :: [Char]

**allowedTypes** :: [[Char]]

data **GUI**

main GUI type

### Constructors

**GUI**

**mainWin :: Window**
main window

**entryFrom :: Entry**
text entry for m_From

**entryTo :: Entry**
text entry for m_To

**entryCC :: Entry**
text entry for m_Cc

**entrySubject :: Entry**
text entry for m_Subject

**entrySearch :: Entry**
text entry for searching the db

**wid_button1 :: Button**
button 'new mail'

**wid_button3 :: Button**
button 'import pop3'

**wid_button4 :: Button**
button 'import mbox'

**wid_button5 :: Button**
button 'remove mail'

**wid_button7 :: Button**
button 'save/forward mail'

**wid_button8 :: Button**
button 'send/reply mail'

**wid_button2 :: Button**
button search

**wid_buttonbox3 :: ButtonBox**
: button box for buttons 1-8

**wid_linkButton :: LinkButton**
: link button for attachment uri

**wid_treeviewMails :: TreeView**
: treeview `mails`

**wid_treeviewAttachments :: TreeView**
: treeview `mailparts`

**wid_treeviewLabels :: TreeView**
: treeview `labels`

**mwTextView :: TextView**
: textview for mailbody

**widStatusBar :: Statusbar**
: statusbar

**statusbarContextId :: ContextId**
: statusbar context

**appMode :: IORef String**
: application mode: empty, `send`, or `reply`

**ioMailId :: IORef String**
: last selected mail UUID

---

`data` **LabelInfo**

Type for treeview-store

### Constructors

**LabelInfo**

**lName :: String**

**lCount :: String**

---

# mailcdb-1.0: The couchDB-Mail Package

the couchdb based client implemented in haskell for search and retrieval of e-mail

## Modules

# DB.CDBHandler

Safe Haskell None

This modul handels connections to Apache CouchDB at a high level. It provides functions to conveniently manipulate Documents, views and indexViews created for Apache Lucene in a couchDB.

Contents



## Database functions

**createMyDB** :: IO ()

## Document functions

**updateDocument**

```
:: Data a
=> DB                          database name
-> IDString                    document UUID
-> a                           new document
-> IO (Either IOError Bool)
```

Update couchdb document

return Right True on success; return Left errorString on error

**deleteDocument**

```
:: DB       database name
-> Doc      document id
-> IO ()
```

Delete couchdb document identified by document id

**storeDocument**

```
:: Data a
=> DB             database name
-> a              document
-> IO IDString    UUID of created document
```

Write couchdb document, return written UUID

**retrieveDocument**

```
:: Data a
=> DB                              database name
-> IDString                        document UUID
-> IO (Either ErrorString a)       document
```

Retrieve couchdb document by given id

return Right Doc on success; return Left errorString on error

---

**retrieveDocFromView**

```
:: Data a
=> DB                                database name
-> Doc                               document id
-> ViewString                        index view
-> QueryPairs                        query parameters
-> IO (Either ErrorString [a])
```

Retrieve couchdb document from db on view

return Right Doc on success; return Left errorString on error

---

**retrieveIdDocFromView**

```
:: Data a
=> DB                                database name
-> Doc                               document id
-> ViewString                        index view
-> QueryPairs                        query parameters
-> IO (Either ErrorString [(Doc, a)])
```

Retrieve (id,doc) from db on view

return Right Doc on success; return Left errorString on error

---

**storeJSObject**

```
:: DB            database name
-> JSValue       JSObject
-> IO IDString   UUID of created document
```

Write JSObject to couchdb, return written UUID

## Query functions

---

**queryDB**

```
:: DB                        database name
-> ViewString                index view
-> [(String, String)]        search parameters
-> IO [QueryViewResult]      search results array
```

query cdb with given search parameters, return search results array

---

**queryDBKeys**

| | |
|---|---|
| `:: DB` | database name |
| `-> ViewString` | index view |
| `-> [(String, String)]` | search parameters |
| `-> IO [IDString]` | search results array |

query cdb with given search parameters, return keys only

## Helper functions

**`cleansJSValue :: JSValue -> JSValue`**

removes id from JSValue (new id is provided by db)

## Type synonyms

**`type IDString = String`**

**`type QueryViewResult = (Doc, JSValue)`**

**`type ErrorString = String`**

## Predefined strings

**`designdocAccount :: Doc`**

**`designdocMail :: Doc`**

**`mailDB :: DB`**

**`mailDBString :: [Char]`**

# DB.CDBHelper

Safe Haskell None

This modul creates design docs in couchDB. Javascript functions for map, reduce, fti, and validation functions are defined as ViewMap, ViewMapReduce, FTIFunction and ValidFunction constuctors. These elements are encapsulated in a CouchDB computation and run by runCouchDB'

## Documentation

**setMailViews**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create views with defined map functions for mails

**setAccountViews**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create views with defined map functions for accounts

**setFTI**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create design doc for index functions

**setValidation**

```
:: String   db string
-> String   designdoc string
-> IO ()
```

create design doc for validation function

# DB.DAL

Safe Haskell None

Database Abstraction Layer: provides type safe functions for accessing couchdb

Contents



## Document functions

**prepareDB** :: IO ()

Create database, design documents for mail, accounts, full text search, and validation and store sample mail account

**storeMail**

:: Mail — mail
-> IO IDString — UUID of created document

Store mail to cdb

**storeMailInc**

:: Mail — mail
-> IO ()

Store mail to cdb and increment countRetr in accountDB

**storeAccount**

:: Account — account to store
-> IO IDString — UUID of created document

Store pop3Account to cdb

**retrieveHeader**

:: IDString — UUID of part to retrieve
-> IO (Either ErrorString Header) — errorstring or retrieved part

Retrieve mailpart by given id

return Right Part on success; return Left ErrorString on error

**retrieveAttachments**

:: IDString — UUID of attachment to retrieve
-> IO (Either ErrorString [Attachment]) — errorstring or retrieved attachment

Retrieve mail-attachments by given id

return Right list of attachments on success; return Left ErrorString on error

**updateHeader**

```
:: IDString                        UUID of mailpart to update
-> Header                          updated instance of mailpart to store
-> IO (Either IOError Bool)        errorstring or True
```

Update mailpart

return Right True on success; return Left ErrorString on error

---

**deleteMail**

```
:: IDString    UUID of mailpart to delete
-> IO ()
```

Delete mailpart with given UUID

---

**enableMailAccount**

```
:: Label    label of mail account to enable
-> IO ()
```

enable mail account

---

**disableMailAccount**

```
:: Label    label of mail account to disable
-> IO ()
```

disable mail account

---

**mailAccountStatus**

```
:: Label    label name
-> Bool     activation status
-> IO ()
```

set activation status of mail accounts in cdb (enable or disable)

## Query view functions

---

**retrieveEnabledMailAccounts** `:: IO (Either String [Account])`

Retrieve enabled mail accounts (p_enabled=true)

return Right [Account] on success; return Left ErrorString on error

---

**findAccountsByLabel**

```
:: Label                                    label name
-> IO (Either ErrorString [(Doc, Account)])
```

Retrieve account with given label name

return Right [ID, Account] on success; return Left ErrorString on error

---

**retrieveMailInfoRec**

```
:: Label                                  label name
-> IO (Either ErrorString [MailInfo])     errorString or list of mailInfo
```

Retrieve mailInfo of received mails with given label name (field mailaddress = m_From)

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveMailInfoSent**

```
:: Label                                  label name
-> IO (Either ErrorString [MailInfo])     errorString or list of mailInfo
```

Retrieve mailInfo of outgoing or sent mails with given label name (field mailaddress = m_To )

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveMailInfo**

```
:: IDString                               UUID of mailInfo to retrieve
-> IO (Either ErrorString [MailInfo])     errorString or list of mailinfo
```

Retrieve mailInfo by given id

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveLabels** `:: IO [(String, Int)]`

Retrieve name and number of existing labels from cdb

---

**retrieveAccounts**

```
:: IO (Either ErrorString [Account])    errorString or list of accounts
```

Retrieve all accounts from cdb

return Right [Account] on success; return Left ErrorString on error

## Query full-text-search functions

---

**queryFulltext**

```
:: String                 querytext
-> IO [QueryViewResult]   result of query queryFulltext queryText = queryDB mailDB header [(q, queryText)] carefull!!!
```

fulltext-query DB for queryText

---

**queryFulltextKeys**

```
:: String          querytext
```

-> IO [IDString] key only result queryFulltextKeys queryText = queryDBKeys mailDB header [(q, queryText)] carefull!!!

fulltext-query DB for queryText, return keys only

---

```
type Label = String
```

## Helper functions

---

**incrementCountRetr** :: Label -> IO ()

Increment number of retrieved mails (countRetr) in cdb

---

**addAttachment2JSON** :: JSValue -> [Attachment] -> JSValue

Add attachment to JSON Object

---

```
type IDString = String
```

---

```
type ErrorString = String
```

---

**maildbURL** :: [Char]

---

# Mail.MBOX

Safe Haskell None

Mbox handling functions

Example of opening a mbox file:

```
module Main where
import ...

main :: IO ()
main = do
    let filePath = sample1.mbox
    mails <- doImportMbox filePath
    sample code
    sample code
    sample code
```

## Documentation

**doImportMbox**

```
:: FilePath                      file path
-> IO (Either String [Mail])     errorString or list of mails
```

**splitMailString**

```
:: String      MBOX string
-> [String]    list of mail strings
```

Parse MBOX string, separate different mail strings

# Mail.POP3

Safe Haskell None

This module handels connections to POP3-Server

Example of downloading messages from a pop3 server:

```
module Main where
import ...

mailAccount = POP3Account {
                    p_host      = pop.example.com,
                    p_user      = user@example.com,
                    p_pass      = password,
                    p_email     = user@example.com,
                    p_label     = label,
                    p_countRetr = 100,
                    p_enabled    = True }

main :: IO ()
main = do
    mails <- retrieveMailFromAccount mailAccount
    sample code
    sample code
    sample code
```

Contents



## Datatypes

data **POP3Handler**

Datatype encapsulating all IO functions for pop3 handling

### Constructors

**POP3Handler**

**ph_conn :: IO ()**
open connection to pop3 server

**ph_disconn :: IO ()**
close connection to pop3 server

**ph_stat :: IO Int**
return total number and size of emails on server

**ph_list :: IO ()**
print size of 1st email

**ph_new :: IO Int**
return number of new emails on server

**ph_fetch :: Int -> IO [ByteString]**
fetch emails starting at given number to total number of emails on server

**connectToPopServerWith**

:: Account the POP3 account to be queried

| | |
|---|---|
| `-> IO POP3Handler` | return POP3Handler |

Constructor of `POP3Handler` holding implementations of IO

## Mail Handling Functions

**`retrieveMailFromAccount`**

| | |
|---|---|
| `:: Account` | pop3Account |
| `-> IO (Either String [Mail])` | return Right [[Mail]] on successful retrieval, Left ErrorString on error |

Convert mime-messages returned from `getNewMessages` to Types.Mail, add label (for categorizing) to mails.

return Right constructor with list of mails on success; return Left constructor with errorString on error

---

**`getNewMessages`**

| | |
|---|---|
| `:: Account` | pop3Account holding pop3 login info and # of retrieved mails |
| `-> IO [Message]` | new messages as MIME-Message |

Initiate pop3handler with given pop3account, connect, fetch mail byte stream of new mails (according to pop3Account), parse it and return as MIME-message.

Errors trigger IO exception.

---

# Mail.SMTP

Safe Haskell None

Sending mailparts via SMTP

## Documentation

**doSendMimeMail**

```
:: Header                          mail-header
-> String                         path to attachment
-> String                         hostname of smtp server
-> String                         mailaddress of sender
-> IO (Either IOError Bool)
```

Send mime-mail with given mailpart using SMTP

return Right True on success; return Left constructor with errorString on error

# Mailcdb

Safe Haskell None

Top-level code

Implements high level functions to manipulate mails stored in db and to import new mails to db.

Contents



## Mail manipulating functions

**runNewMail**

```
:: IO IDString
```
IDString of created mailpart

Create new empty mailpart and store in db

return IDString of created mailpart

**runNewMailReply**

```
:: IDString                                          IDString of mailpart to cite in new mailpart
-> IO (Either String (Header, IDString))
```

Create mail-reply by citing existing mailpart and store in db

return Left ErrorString on error; return Right mailpart and IDString of newly created mailpart as tuple

**runNewMailForward**

```
:: IDString                                          IDString of mailpart to cite in new mailpart
-> IO (Either String (Header, IDString))
```

Create mail-forward by citing existing mailpart and store in db

return Left ErrorString on error; return Right mailpart and IDString of newly created mailpart as tuple

**runSendMail**

```
:: IDString                                          IDString of mailpart to send
-> IO (Either ErrorString LabelString)               Errorstring or name of assigned label
```

Send body of mailpart specified by IDString as mail

return Left ErrorString when mail can not be retrieved from db, when insufficient mail entries are set, or when an invalid server hostname is given in configfile; return Right outbox-labelname when mail cannot send due to smtp or connection problems; return Right sent-labelname when mail was successfully transmitted to smtp server

**runDeleteMail**

```
:: IDString
```
IDString of parent mailpart to delete

```
-> IO ()
```

Remove parent mailpart and associated children-mailparts from db

## Mail importing functions

**runImportMboxFrom**

```
:: FilePath    filepath to mbox file
-> IO ()
```

Open mbox-file with given filepath, convert containing Message-Strings to Types.Mail.Part, store to db

**runRetrieveMailFromActivePopAccounts** `:: IO ()`

Retrieve all enabled POP3 accounts from db, query POP3 server, retrieve mails, and store them to db

## Helper functions

**labelNameSent** `:: [Char]`

label-name for sent mails

**labelNameDrafts** `:: [Char]`

label-name for draft-mails

**labelNameOutbox** `:: [Char]`

label-name for mails to be sent

**runConditionFunction** `:: Show t => IO (Either t t1) -> IO a -> (t1 -> IO ()) -> IO ()`

run functions depending on return of conditionFunction: success (function runOnRight) or fail (function runOnLeft)

**runConditionFunction'** `:: Show t => IO (Either t [t1]) -> IO a -> (t1 -> IO ()) -> IO () -> IO ()`

run functions depending on return of conditionFunction: success (function runOnRight) or fail (function runOnLeft) or emptyList

## DB.DAL call-through functions

### Document functions

**prepareDB** `:: IO ()`

Create database, design documents for mail, accounts, full text search, and validation and store sample mail account

**storeMail**

```
:: Mail          mail
```

```
-> IO IDString
```
UUID of created document

Store mail to cdb

---

**storeMailInc**

```
:: Mail
```
mail

```
-> IO ()
```

Store mail to cdb and increment countRetr in accountDB

---

**storeAccount**

```
:: Account
```
account to store

```
-> IO IDString
```
UUID of created document

Store pop3Account to cdb

---

**retrieveHeader**

```
:: IDString
```
UUID of part to retrieve

```
-> IO (Either ErrorString Header)
```
errorstring or retrieved part

Retrieve mailpart by given id

return Right Part on success; return Left ErrorString on error

---

**retrieveAttachments**

```
:: IDString
```
UUID of attachment to retrieve

```
-> IO (Either ErrorString [Attachment])
```
errorstring or retrieved attachment

Retrieve mail-attachments by given id

return Right list of attachments on success; return Left ErrorString on error

---

**updateHeader**

```
:: IDString
```
UUID of mailpart to update

```
-> Header
```
updated instance of mailpart to store

```
-> IO (Either IOError Bool)
```
errorstring or True

Update mailpart

return Right True on success; return Left ErrorString on error

---

**deleteMail**

```
:: IDString
```
UUID of mailpart to delete

```
-> IO ()
```

Delete mailpart with given UUID

---

**enableMailAccount**

```
:: Label    label of mail account to enable
-> IO ()
```

enable mail account

---

**disableMailAccount**

```
:: Label    label of mail account to disable
-> IO ()
```

disable mail account

---

**mailAccountStatus**

```
:: Label    label name
-> Bool     activation status
-> IO ()
```

set activation status of mail accounts in cdb (enable or disable)

## Query view functions

---

**retrieveEnabledMailAccounts** `:: IO (Either String [Account])`

Retrieve enabled mail accounts (p_enabled=true)

return Right [Account] on success; return Left ErrorString on error

---

**findAccountsByLabel**

```
:: Label                                    label name
-> IO (Either ErrorString [(Doc, Account)])
```

Retrieve account with given label name

return Right [ID, Account] on success; return Left ErrorString on error

---

**retrieveMailInfoRec**

```
:: Label                                  label name
-> IO (Either ErrorString [MailInfo])     errorString or list of mailInfo
```

Retrieve mailInfo of received mails with given label name (field mailaddress = m_From)

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveMailInfoSent**

```
:: Label                                  label name
-> IO (Either ErrorString [MailInfo])     errorString or list of mailInfo
```

Retrieve mailInfo of outgoing or sent mails with given label name (field mailaddress = m_To )

return Right [MailInfo] on success; return Left ErrorString on error

**retrieveMailInfo**

```
:: IDString                                  UUID of mailInfo to retrieve
-> IO (Either ErrorString [MailInfo])       errorString or list of mailinfo
```

Retrieve mailInfo by given id

return Right [MailInfo] on success; return Left ErrorString on error

---

**retrieveLabels** `:: IO [(String, Int)]`

Retrieve name and number of existing labels from cdb

---

**retrieveAccounts**

```
:: IO (Either ErrorString [Account])   errorString or list of accounts
```

Retrieve all accounts from cdb

return Right [Account] on success; return Left ErrorString on error

## Query full-text-search functions

---

**queryFulltext**

```
:: String                  querytext
-> IO [QueryViewResult]    result of query queryFulltext queryText = queryDB mailDB header [(q, queryText)] carefull!!!
```

fulltext-query DB for queryText

---

**queryFulltextKeys**

```
:: String          querytext
-> IO [IDString]   key only result queryFulltextKeys queryText = queryDBKeys mailDB header [(q, queryText)] carefull!!!
```

fulltext-query DB for queryText, return keys only

---

`type` **Label** `= String`

## Helper functions

---

**incrementCountRetr** `:: Label -> IO ()`

Increment number of retrieved mails (countRetr) in cdb

---

**addAttachment2JSON** `:: JSValue -> [Attachment] -> JSValue`

Add attachment to JSON Object

---

`type` **IDString** `= String`

```
type ErrorString = String
```

```
maildbURL :: [Char]
```

---

# Main

Safe Haskell None

Program entry point

## Documentation

**main** :: IO ()

Run mailcdb with given list of arguments. When no arguments are given mailcdb starts with gtk-gui.

```
Usage: mailcdb gui
                    Start mailcdb with gtk-gui
       mailcdb db prepare
                    Create new db in couchdb, create predefined
                    design documents for couchdb views, full
                    text search, and validation
       mailcdb account new
                    Create new pop3 mail account
       mailcdb account list
                    List pop3 mail accounts
       mailcdb account enable <accountLabel>
                    Enable pop3 mail account with given label
       mailcdb account disable <accountLabel>
                    Disable pop3 mail account with given label
       mailcdb mail new
                    Create new Mail
       mailcdb mail fetch
                    Fetch mails from enabled pop accounts
       mbox <filePath>
                    Import mails from given mbox file
       mailcdb help
                    Display usage info
```

# Types.Account

Safe Haskell: Safe-Inferred

Types that are commonly used in mailcdb

## Documentation

```
data Account
```

Record which contains all information to connect and authenticate to a POP3 server and stores the number of retrieved emails

### Constructors

**POP3Account**

**p_host :: String**
: IP or hostname of pop3 server

**p_user :: String**
: username of valid email account

**p_pass :: String**
: password of valid email account

**p_email :: String**
: full email address

**p_label :: String**
: label to be used for storing emails of this account

**p_countRetr :: Int**
: number of mails retrieved so far

**p_enabled :: Bool**
: switch for enabling or disabling this account

### Instances

Eq Account
Data Account
Ord Account
Read Account
Show Account
Typeable Account

**samplePOP3Account :: Account**

sample mail account

# Types.Mail

Safe Haskell None

Types that are commonly used in mailcdb

## Documentation

```
type LabelString = String
```

```
data Mail
```

A mail consists of a header and a list of attachments

### Constructors

**Mail**

**m_header :: Header**
Mailheader

**m_parts :: [Attachment]**
List of attachments

### Instances

Show Mail

```
data Header
```

Mail header represented as algebraic data type, holding header information

### Constructors

**Header**

**h_from :: String**
From header

**h_body :: String**
body

**h_to :: String**
recipient

**h_cc :: String**
cc field

**h_label :: [String]**
list of labels

**h_sender :: String**
sender

**h_date :: String**
sender date

**h_mime :: String**

mime version

**h_subject :: String**
subject of mail

**h_contentType :: String**
content type

**h_contentTypeParameter :: String**
content type parameter

**_id :: String**
uuid in couchdb

### Instances

Eq Header
Data Header
Ord Header
Read Header
Show Header
Typeable Header

---

data **Attachment**

Type representing attachments of mails

### Constructors

**Attachment**

**a_filename :: String**
filename of attachment

**a_type :: String**
type of attachment

**a_content :: String**
body

**a_length :: Int**
length of body

### Instances

Eq Attachment
Data Attachment
Ord Attachment
Read Attachment
Show Attachment
Typeable Attachment

---

data **MailInfo**

This type provides minimal info about mail

### Constructors

**MailInfo**

**mId :: String**
uuid in couchdb

**mAddress :: String**
mail address (to or from)

**mSubj :: String**
subject

**mDate :: String**
sender date

**mContentType :: String**
content type

### Instances

Data MailInfo

Show MailInfo

Typeable MailInfo

---

**emptyHeader** :: Header

Construct empty header type

---

**writeLabel** :: String -> Mail -> Mail

Add given string as label to mail

---

**convertMessageToMail**

:: Message — mime-mail message (Codec.MIME.String.Message) to be converted

-> Mail — returned mail message of type Types.Mail

Convert mail message of type Codec.MIME.String.Message to type Types.Mail

---

**incrementFilenames**

:: Mail

-> Mail

make filenames of attachments unique

---

**parseInfo**

:: MessageInfo

-> Header — mail message with parsed info

-> Header — returned mail message of type Types.Header

Parse MessageInfo of mime-mail and write to type Types.Header

**parseContent**

```
:: MessageContent
-> Header
-> Mail                 returned list of mail messages
```

Parse MessageContent of mime-mail and write to mail message of type Types.Mail

when content is some sort of multipartcontent function `convertMessageToMail` is called recursivly and multipart-mails are attached to returned list of mails

---

**parseAttachment**

```
:: [Message]
-> [Attachment]    returned list of attachments
```

Parse parts of mime-mail and return list of type Types.Attachment

---

**month** `:: [[Char]]`

Convert textual presentation of month to numeric

---

**month2Numeric** `:: Month -> String`

---

# UI.CLI

Safe Haskell None

This module provides a command line interface for testing some functions

## Documentation

**readmeFile** :: [Char]

**runCLI** :: IO ()

Main function for using the command line interface

**cliListAccounts** :: IO ()

List all accounts in db

**cliNewAccount** :: IO ()

Store new account to db

**cliNewMail** :: String -> String -> String -> IO ()

Create new mail and store to db

**enterMailInfo** :: IO [String]

Collect mail information from inputs

**main_example** :: IO ()

Store new account, enable it, retrieve messages, and store them do db

# UI.GUI

Safe Haskell None

This module provides a GTK user interface

## Documentation

**runGUI** :: IO ()

Main GTK+ loop

**loadGlade**

:: FilePath filepath of glade file
-> IO GUI

Load and define all gtk elements

**connectGui** :: GUI -> IO (ConnectId Button)

Setup all event handler for GTK+ elements

# UI.Helper

Safe Haskell None

action handler for buttons

Contents



## Widget action handler

**buttonReply**

```
:: GUI
-> ListStore LabelInfo
-> ListStore MailInfo
-> IDString
-> IO ()
```

Action handler for button `reply`: create new mail-reply and select it

**buttonImportPop**

```
:: GUI
-> IO ()
```

Action handler for button 'import pop': start importing mails of all stored mail accounts, when finished change button label to `refresh`

**buttonForward**

```
:: GUI
-> ListStore LabelInfo
-> ListStore MailInfo
-> IDString
-> IO ()
```

Action handler for button `forward`: create new mail-reply and select it

**buttonSend**

```
:: GUI
-> ListStore MailInfo
-> ListStore LabelInfo
-> IO ()
```

Action handler for button `send`: start sending an email

**buttonRemove**

```
:: GUI
-> ListStore MailInfo
-> ListStore LabelInfo
-> ListStore a

-> IO ()
```

Action handler for button `remove`: delete selected mail from db

---

**buttonNew1**

```
:: GUI
-> ListStore LabelInfo
-> ListStore MailInfo
-> IO ()
```

Action handler for button `new`: store new mail to db and select it

---

**buttonSave**

```
:: GUI
-> ListStore MailInfo
-> ListStore LabelInfo
-> IO ()
```

Action handler for button `save`: save data from mail input widgets to db

---

**buttonImportMbox**

```
:: GUI
-> FilePath
-> IO ()
```

Action handler for button 'import mbox': start importing mails from mbox, when finished change button label to `refresh`

---

**buttonSearch**

```
:: GUI
-> ListStore MailInfo
-> IO [()]
```

Action handler for button `search`: get text from text entry, start db query, return results to treeview `mails`

---

**openOpenFileChooser**

```
:: Window
-> IO (Maybe FilePath)
```

Open file chooser dialog for file opening (select mbox)

## TreeView action handler

---

**treeviewAttachmentsSelectionChanged**

```
:: ListStore Attachment
-> t
-> GUI
-> IO ()
```

Handler for event `selection_changed` of treeview `attachments`

---

**treeviewLabelsSelectionChanged**

```
:: GUI
-> ListStore LabelInfo
-> t
-> ListStore MailInfo
-> ListStore a
-> IO ()
```

Handler for event `selection_changed` of treeview `labels`: adjust button sensitivities, set appState according to selected label, update treeview `mails`

---

**treeViewMailSelectionChanged**

```
:: ListStore MailInfo
-> GUI
-> ListStore Attachment
-> t
-> IO ()
```

Handler for event `selection_changed` of treeview `mails`: adjust button sensitivities, update treeview `attachments`

---

**populateTreeviewLabels**

```
:: ListStore LabelInfo
-> IO ()
```

Populate info to treeview `labels`

## Set widget attributes

---

**setTreeviewAtributes**

```
:: (TreeViewClass self, TreeModelClass (model row), TypedTreeModelClass model)
=> self
-> model row
-> String
-> Int
-> (row -> [AttrOp CellRendererText])
-> IO ()
```

Set several common attributes for treeview widgets

---

**setMailDetailsEditable**

```
:: GUI
-> Bool
-> IO ()
```

Set widget's editable flag (r or rw-mode)

---

**pushStatusText**

```
:: GUI
-> String
-> IO ()
```

Push given text to statusbar

---

**emptyMailDetail**

```
:: GUI
-> IO ()
```

Empty mail details widgets

## Helper functions

---

**retrieveLabels** `:: IO [(String, Int)]`

Retrieve name and number of existing labels from cdb

---

**convertToLabelInfo**

```
:: [(String, Int)]
-> [LabelInfo]
```

Convert list of pairs to type LabelInfo

---

# UI.Types

Safe Haskell None

Types that are commonly used in UI.* Modules

## Documentation

**gladeFile** :: [Char]

**allowedTypes** :: [[Char]]

data **GUI**

main GUI type

### Constructors

**GUI**

**mainWin :: Window**
main window

**entryFrom :: Entry**
text entry for m_From

**entryTo :: Entry**
text entry for m_To

**entryCC :: Entry**
text entry for m_Cc

**entrySubject :: Entry**
text entry for m_Subject

**entrySearch :: Entry**
text entry for searching the db

**wid_button1 :: Button**
button 'new mail'

**wid_button3 :: Button**
button 'import pop3'

**wid_button4 :: Button**
button 'import mbox'

**wid_button5 :: Button**
button 'remove mail'

**wid_button7 :: Button**
button 'save/forward mail'

**wid_button8 :: Button**
button 'send/reply mail'

**wid_button2 :: Button**
button search

**wid_buttonbox3 :: ButtonBox**

button box for buttons 1-8

**wid_linkButton :: LinkButton**

link button for attachment uri

**wid_treeviewMails :: TreeView**

treeview `mails`

**wid_treeviewAttachments :: TreeView**

treeview `mailparts`

**wid_treeviewLabels :: TreeView**

treeview `labels`

**mwTextView :: TextView**

textview for mailbody

**widStatusBar :: Statusbar**

statusbar

**statusbarContextId :: ContextId**

statusbar context

**appMode :: IORef String**

application mode: empty, `send`, or `reply`

**ioMailId :: IORef String**

last selected mail UUID

---

`data` **LabelInfo**

Type for treeview-store

### Constructors

**LabelInfo**

**lName :: String**

**lCount :: String**

---