# Entwurfsbeschreibung

16. Mai 2011

## 1 Allgemeines

Buddycloud ist ein verteiltes soziales Netz auf der Basis des Extensible Messaging and Presence Protocols, welches das bestehende XMPP-Netzwerk durch eine Reihe von Protokollerweiterungen um soziale Netzwerkkomponenten erweitert, gleichzeitig aber die bestehende föderierte Infrastruktur weiternutzt.

## 2 Produktübersicht

Buddycloud erlaubt es Nutzern sogenannte Channels zu erzeugen, den Channeln anderer Nutzer beizutreten und Nachrichten auf diesen Channeln zu publizieren. Hierbei kann eine Vielzahl von Clients genutzt werden. Ein klassisches Webfrontend bietet Nutzern die Möglichkeit einen vollständigen XMPP-Client im Browser auszuführen und am Netzwerk teilzunehmen ohne Software auf dem lokalen System zu installieren. Da das Netz von Channelservern föderiert ist kann jeder mit den entsprechenden technischen Kenntnissen einen eigenen Channelserver betreiben um seine persönlichen Daten zu schützen oder auch einem bekannten Channelserver mit seinem existierenden XMPP-Account benutzen.

## 3 Grundsätzliche Struktur- und Entwurfsprinzipien für das Gesamtsystem

Buddycloud stellt eine von vielen Erweiterungen des XMPP-Netzwerks dar. Das Netz ist vollständig föderiert, das heisst Nutzer kommunizieren mit Servern und Server kommunizieren untereinander um das Gesamtnetz zu bilden. Server finden sich gegenseitig über DNS; dementsprechend erfolgt eine Zuordnung von Nutzern zu Hosts nach dem Schema *nutzer@host.tld*. Eine solche eindeutige Identifikation wird JID genannt.

Erweiterungen des XMPP-Protokolls werden im Normalfall nicht in den einzelnen Servern direkt implementiert sondern in externe Komponenten ausgelagert welche über das *Jabber Component Protocol* mit dem Server interagieren. Komponenten auf einer XMPP-Serverinstanz werden über den Präfix des Hostnamens adressiert (*conferences.host.tld* wird vom Server an die Multiuserchatkomponente geleitet), wobei der eigentliche XMPP-Server dabei nur als ein Vermittler zwischen Client und Komponente agiert. Da XMPP XML-basiert ist, können Erweiterungen einfach als neue Namensräume in das bestehende Ökosystem integriert werden. Nachrichten in Buddycloud sind

einfache im *Atom Syndication Format* gehaltene und *Atom Activity Extensions* nutzende XML-Dokumente welche direkt in den XMPP-Nachrichten eingebettet werden. Serverseitig besteht Buddycloud aus dem Channelserver der das Channelprotokoll, welches alle Aspekte der Interaktion zwischen Channeln und Nutzern definiert, implementiert. Das Buddycloud Channelprotokoll basiert zum überwiegenden Teil auf der in *XEP-0060* spezifizierten Publish/Subscribe-Funktionalität.

## 3.1 Publish/Subscribe (XEP-0060)

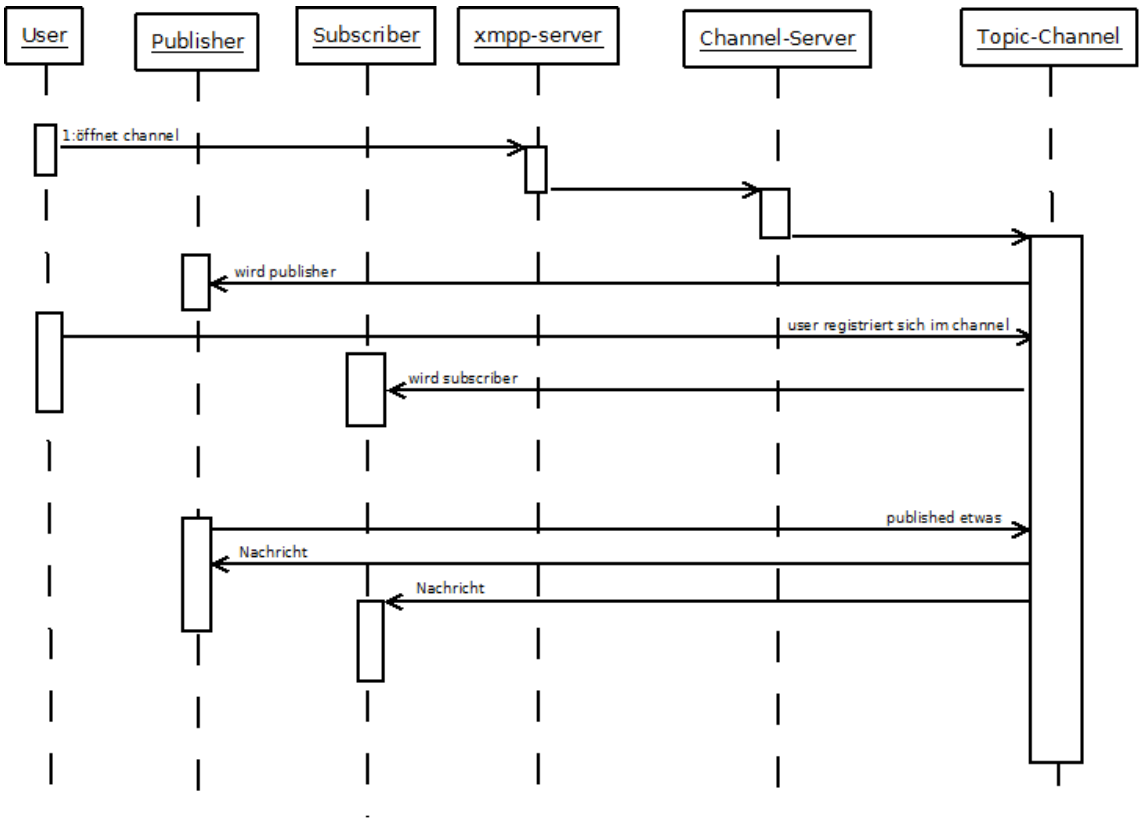



Abbildung 1: Publish-Subscribe mit Buddycloudterminologie

Die XMPP-Erweiterung *Publish/Subscribe* ist weitestgehend mit dem weitverbreiteten Observer-Entwurfsmuster identisch und dient hauptsächlich der Verbesserung der Effizienz und Benutzerfreundlichkeit im Vergleich zu klassischen pollenden Ansätzen. Die Erweiterung erlaubt es XMPP-Entitäten Pub/Sub-Knoten auf einem Pub/Sub-Service zu erzeugen, Knoten zu abonnieren und Daten auf diesen publizieren; im Gegensatz zu auf konstantem Pollen basierenden Protokollen, bei denen in gewissen Abständen Änderungen manuell abgefragt werden müssen, werden bei Pub/Sub alle Abonnenten[1] eines Knotens automatisch über neue Inhalte informiert. Ein Channel in Buddycloudterminologie ist nichts anderes als ein Pub/Sub-Knoten (mit den entsprechenden Erweiterungen und Änderungen die das Channelprotokoll mitbringt).

[1] In Pub/Sub-Terminologie: Subscriber

## 3.2 BOSH (XEP-0206 und XEP-0124)

*XMPP über BOSH* ermöglicht es, XMPP Streams mittels *HTTP Anfrage/Antwort Paaren* zu übertragen. Es wird für die Kommunikation zwischen *Webclient* und dem XMPP-Netzwerk genutzt, das heisst es ermöglicht XMPP über HTTP zu sprechen und dementsprechend einen vollständigen XMPP-Client im Browser auszuführen. Dank *BOSH* können beide Seiten zu jeder Zeit beliebige Daten mit geringer Latenz senden und empfangen und eine langlebige Session intakt halten selbst wenn die darunterliegenden TCP-Verbindungen nur kurzlebig sind.

Die grundlegende Funktionsweise besteht darin, dass der Client eine HTTP Anfrage sendet und der Server erst eine Antwort gibt, wenn er selbst Daten senden muss. Damit der Server weiter selbständig Daten an den Client senden kann, wird vom Client eine weitere (womöglich leere) HTTP Anfrage übermittelt, auf die der Server beizeiten antworten kann. Falls der Client derweil weitere Daten an den Server senden muss, erstellt er eine weitere HTTP Anfrage mit diesen Daten. Wenn dabei HTTP Pipelining (d.h. aufeinanderfolgende HTTP Anfragen) nicht möglich ist, wird eine weitere HTTP Verbindung erstellt, um die Daten zu übermitteln. Dabei wechselt sich, welche Verbindung zum Senden vom Client zum Server und welche vom Server zum Client verwendet wird.

# 4 Grundsätzliche Struktur- und Entwurfsprinzipien der einzelnen Pakete

## 4.1 Channelserver

Der Channelserver ist eine in *node.js* geschriebene XMPP-Komponente welche das Channelprotokoll implementiert. Der Quellcode folgt strikt dem Model-View-Controller-Konzept, wobei das Model (*model_couchdb.js* bzw. *model_postgres.js*) den Zugriff auf die zu nutzende Datenbank abstrahiert. Zur Zeit können sowohl die dokumentenorientierte *Apache CouchDB* als auch die relationelle Datenbank *PostgreSQL* genutzt werden. Der View (*xmpp_pubsub.js*) stellt Daten aus dem Model dar, d.h. er enkapsuliert die Daten in valides XMPP. Hierfür wird eine externe Softwarebibliothek, node-xmpp, genutzt welche die Erzeugung der XML-Struktur deutlich erleichtert und simple Funktionen zum Parsen eingehender Nachrichten bereitstellt. Eingehende XMPP-Nachrichten werden geparst und an den Controller (*controller.js*) weitergereicht, welcher Validierung vornimmt, gegebenenfalls Daten vom Model anfordert und die entsprechenden Funktionen im View aufruft um XMPP-Nachrichten zu erzeugen.

Der *node.js*-Philosophie entsprechend ist die Anwendung hochgradig asynchron. Sämtliche Kommunikation zwischen den Komponenten erfolgt über Callbackfunktionen wodurch jeglicher State übergeben werden muss; anonyme Funktionen und die *step*-Bibliothek werden genutzt um den Quellcode dennoch relativ flach zu halten und tiefe Verschachtelungen zum grossen Teil zu umgehen.

### 4.1.1 Model

Als Datenbankbackend wird hier nur das dokumentenorientierte CouchDB-Interface betrachtet. Da alle Inhalte der Datenbank einzelne JSON[2]-formatierte Dokumente sind, muss jede klassische Abfrage (Selektion) durch Views verarbeitet werden die über die Menge aller Dokumente operieren (*model_couchdb.js* ab Zeile 174). Jeder Channel wird in der Datenbank als ein einzelnes JSON-Dokument abgelegt, dass folgendes Format hat:

[2] JavaScript Object Notation

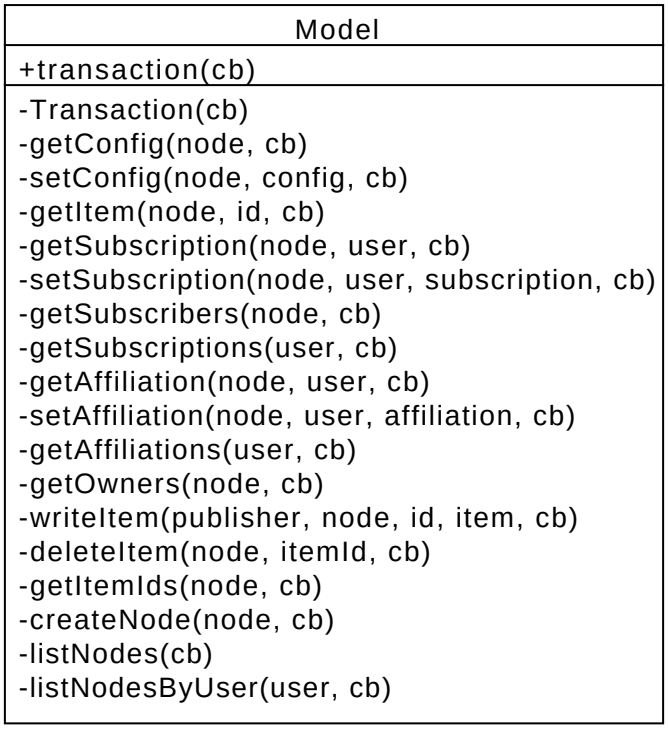



Abbildung 2: CouchDB Model

```
{ _id: '/user/barfoo@localhost/channel',
  _rev: '25-6d217d3dd14e85a8ceb2d6370a349198',
  config:
   { title: 'barfoo\'s node',
     description: 'Where barfoo publishes things',
     type: 'http://www.w3.org/2005/Atom',
     accessModel: 'open',
     publishModel: 'subscribers',
     creationDate: '2011-05-08T13:18:45.842Z' },
  subscriptions: { 'xmpp:foo@localhost': 'subscribed' } }
```

Publizierte Nachrichten werden analog als JSON-Dokument gespeichert:

```
{ _id: '/user/barfoo@localhost/channel&itemID',
  _rev: '25-6d217d3dd14e85a8ceb2d6370a349197',
  date: '2011-05-08T13:18:45.842Z',
  xml: '<entry xmlns="http://www.w3.org/2005/Atom" xmlns:activity="http://
      activitystrea.ms/spec/1.0/">
                <published>2010-01-06T21:41:32Z</published>
                <author>
                   <name>koski@buddycloud.com</name>
                   <jid xmlns="http://buddycloud.com/atom-elements-0">
                       koski@buddycloud.com</jid>
                </author>
                <content type="text">Test</content>
```

```
            <activity:verb>post</activity:verb>
            <activity:object>
              <activity:object-type>note</activity:object-type>
            </activity:object>
          </entry>' }
```

Aufgrund der Dokumentenorientierung ist es nicht möglich einzelne Felder eines Dokuments zu verändern. Es muss immer das gesamte Dokument ausgelesen, abgeändert und dann in die Datenbank zurückgeschrieben werden.

### 4.1.2 View

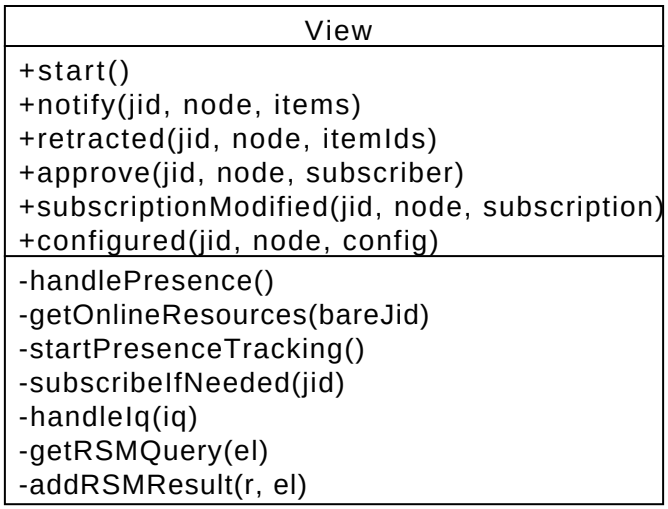



Abbildung 3: View

Der View stellt die Schnittstelle des Channelservers zum XMPP-Netzwerk dar. Eingehende Nachrichten werden nach Typ an entsprechende Unterfunktionen weitergereicht. Ein Grossteil der Logik steckt in der Funktion *handleIq()* welche eingehende Requests vom Typ *IQ*[3] bearbeitet. Falls eine Nachricht als den Channelserver betreffend erkannt wurde, also die entsprechenden Auszeichner und Attribute[4] des Requests existieren, werden die Daten aus dem XML-Dokument extrahiert und mittels der Funktion *controller.request()* an den Controller übergeben. Zusätzlich wird zwingend eine Callbackfunktion definiert die Erfolg oder Versagen der Anfrage mit einer Meldung an den Client übermittelt.

Im View werden zusätzlich die Presenzinformationen aller Subscriber verwaltet. Jeder Client sendet in regelmässigen Abständen und bei Statuswechseln eine Presenznotifikation an alle Mitglieder seines Rosters. Dieses Tracken des Status dient zum einen dazu Subscribernotifikationen an alle eingeloggten Clients eines XMPP-Accounts zu emittieren[5], zum anderen werden Notifikationen nur an JID mit einem eingeloggten Client versandt.

[3] Info/Query
[4] XML Namensraum, IQ-Requesttyp, publish-Auszeichner, etc.
[5] RFC3920 überlässt das Verhalten beim Weiterleiten einer Nachricht bei mehreren eingeloggten Clients dem Server.

Wenn der Controller unabhängig von bestehenden Anfragen, XMPP-Nachrichten versenden will existieren Frontendhooks welche Nachrichten versenden können. Eine Liste der verfügbaren Hooks wird bei der Initalisierung des Views an den Controller übergeben.

### 4.1.3 Controller

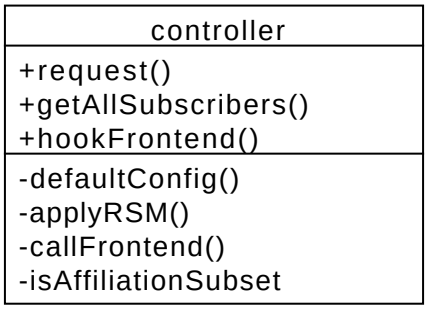



Abbildung 4: Controller

Der Controller (*controller.js*) führt die Requests des Views aus. Hierfür existiert ein Array mit Routinen einer bestimmten Funktionalität. Jede dieser Funktionen unterteilt sich zudem in weitere Unterroutinen die zu verschiedenen Zeitpunkten einer Transaktion ausgeführt werden:

**transaction** ist die Funktion die die eigentlichen Änderungen am Model etc. durchführt. In ihr wird im Normalfall der Grossteil der eigentlichen Backendlogik ausgeführt.

**afterTransaction** wird nach dem Schreiben der durch die Transaktion veranlassten Modeländerungen ausgeführt.

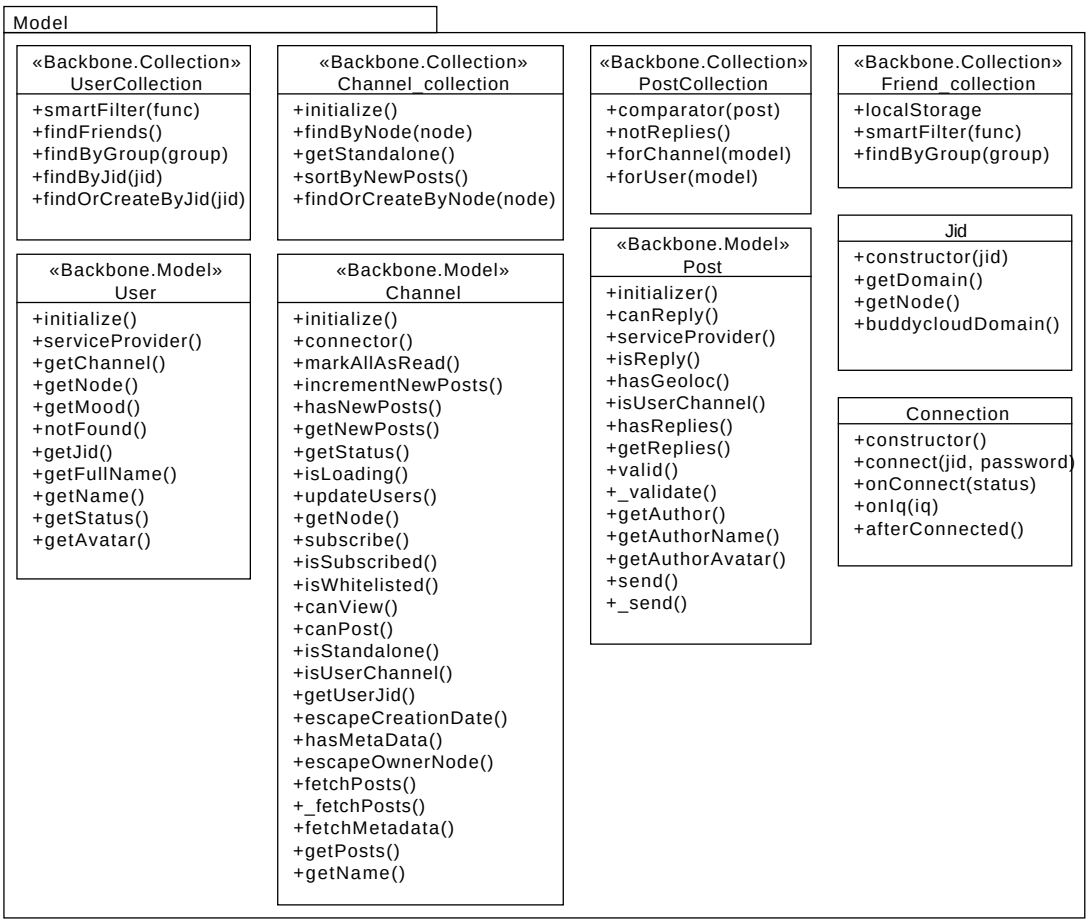
**subscriberNotification** wird letztendlich ausgeführt um alle Subscriber von Änderungen in Kenntniss zu setzen.

Der View emittiert einen Request an den Controller über die Funktion *request()* der dann mittels der Stepbibliothek die Funktionsaufrufe aus dem Array in eine Liste zusammenbaut und grundlegende Informationen aus dem Model ausliest. So werden zum Beispiel vor dem Aufruf der Unterroutine *subscriberNotification* zuerst die JIDs aller Subscriber aus dem Model extrahiert. Nach dem Abarbeiten aller Unterroutinen wird die mit dem Request übergebene Callbackfunktion mit den Ergebnisdaten aufgerufen.

## 4.2 Webclient

Der Webclient basiert auf dem Model-View-Controller-Entwurfsmuster und ist in Coffeescript, einer Sprache die zur Laufzeit in Javascript übersetzt wird, geschrieben. Der Quellcode befindet sich in channel-webclient/app. Coffeescript kennt keine Unterteilung in Pakete, daher ist diese Unterteilung aus der Verzeichnisstruktur und entsprechenden Zusammengehörigkeiten abgeleitet.

### 4.2.1 Model




Das Model repräsentiert die applikationsinternen Daten zusammen mit der zugehörigen Logik. Alle Daten kommen entweder über die Controller oder die Connection ins Model, Inhalte verlassen das Model über die Connection oder werden an die Views weitergerreicht. Das Model beinhaltet folgende Klassen:

**UserCollection** Kapselt eine Menge von Benutzern zu denen der aktuelle Nutzer irgendwie in Kontakt steht, und erlaubt es, diese einfach zu filtern.

**User** Entspricht einem einzelnen, nicht lokalen Nutzer, und erlaubt es, unterschiedliche Informationen, wie Beispielsweise dessen Namen, Status, Avatar oder Channel abzufragen.

**Channel_collection** Ist eine Menge von Channels, auf denen der Benutzer subscribed sein kann, und die z.B. auf neue Posts hin untersucht werden kann.

**Channel** Entspricht einem Knoten auf einem Channel-Server und gibt Aufschluss darüber ob neue Posts vorhanden sind, der entsprechende Knoten eingesehen werden kann, oder ob Nachrichten veröffentlicht werden dürfen.

**PostCollection** Eine PostCollection ist eine Sammlung von Posts, die sowohl für einen Channel als auch für einen User erstellt werden kann. Dies trägt der Tatsache Rechnung, dass sowohl meh-

rere User auf einem Channel als auch ein User auf mehreren Channels Inhalte veröffentlichen können.

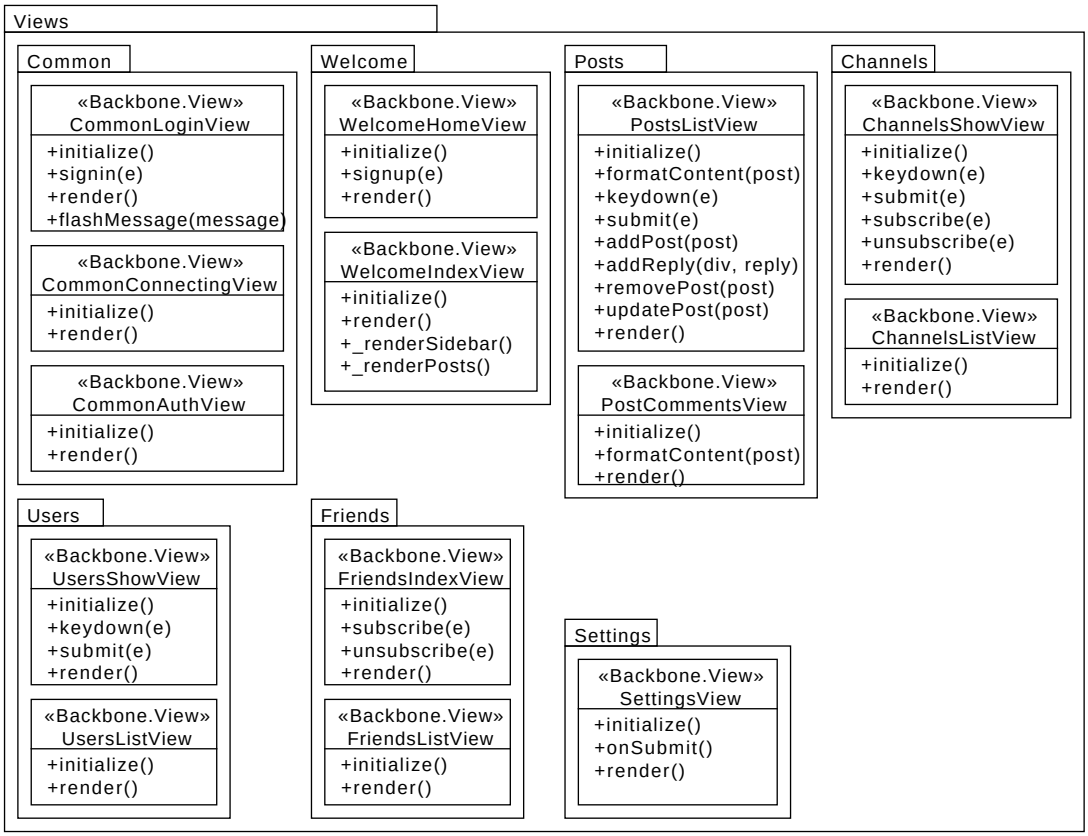
**Post** Ein Post ist eine einzelne Nachricht, die auf einem Channel veröffentlicht wurde. Eine Nachricht kann eine Antwort auf eine andere Nachricht sein, aufschluss über ihren Inhalt und ihren Autor geben, oder mit einer Geolocation versehen sein.

**Friend_collection** Die Friend_collection hilft dabei Kontakte zu filtern.

**Jid** Repräsentiert eine Jid (JabberId) und kann Auskünfte darüber geben.

**Connection** Die Connection ist der Punkt, über den die Webapplikation mit dem Rest des Netzwerks interagiert. Intern verwendet diese Klasse den Connector aus dem Paket Connectors.

### 4.2.2 Views




Innerhalb des Views sind die Klassen wiederum ihren Aufgabenbereichen entsprechend in Pakete unterteilt:

**Common** Kümmert sich um die Anzeige von Loginelementen und dazugehörigen.

**Welcome** Dieses Paket bildet die Hauptansicht des Webclients.

**Posts** Das Posts Paket kümmert sich um das Anzeigen einzelner Posts.

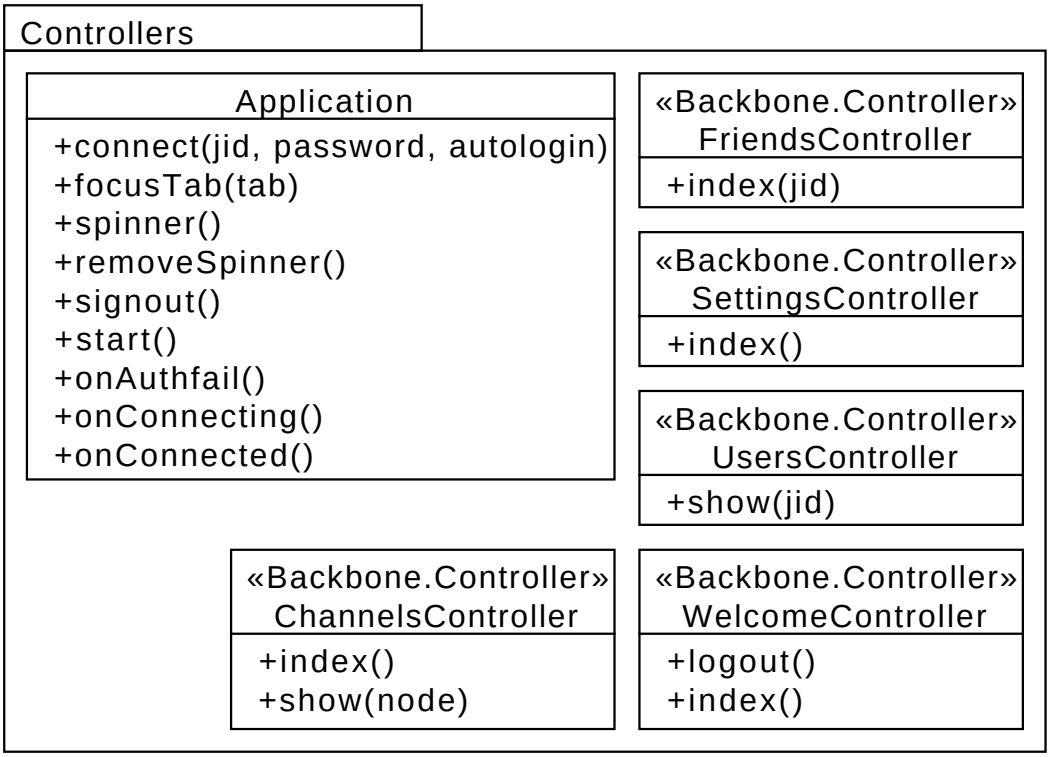
**Channels** Einzelne Channels werden dem Benutzer präsentiert, sodass dieser interagieren kann.

**Users** Das Users Paket stellt Informationen zu anderen Kontakten graphisch dar.

**Friends** Dieses Paket kümmert sich darum, die 'Freunde' des Nutzers zu präsentieren.

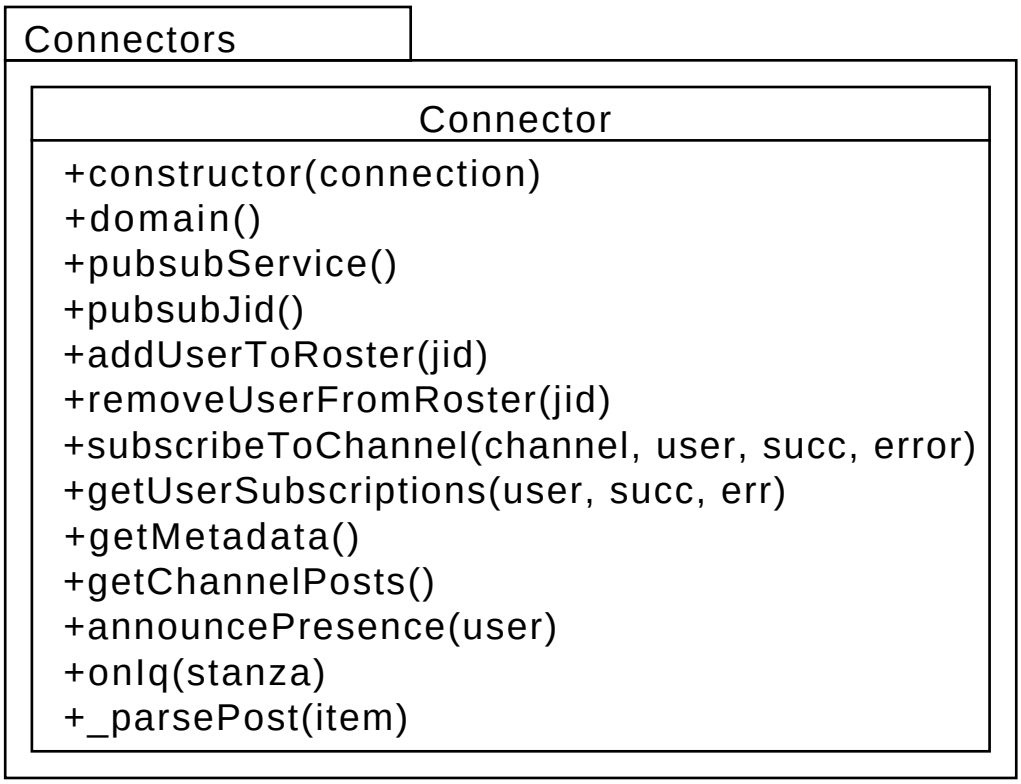
**Settings** Dieses Paket enthält momentan lediglich das SettingsView, die die Nutzereinstellungen präsentiert.

#### 4.2.3 Controllers




Die verschiedenen Klassen des Controllers Paketes bilden Aufgabenbereiche in der Interaktion mit der Website ab. Dazu gehören beispielsweise persönliche Einstellungen vornehmen (Controllers.SettingsController) oder sich Channels anzeigen lassen (Controllers.ChannelController). Die Klassen in diesem Paket warten entsprechend ihrer Aufgaben auf Interaktion des Nutzers und leiten die geforderten Änderungen an das Model weiter.

#### 4.2.4 Sonstiges




Die Klasse Connector wird von Model.Connection verwendet um eine Verbindung zum XMPP-Server des Benutzers herzustellen. Sie ist aber vom MVC-Muster isoliert und in einem eigenen Paket hinterlegt.