Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 6

Frameworkentwicklung, Multi-Plattform

Autor: E. Gailus (B2M)

Co-Autoren: G. Hübsch (CAS)

A. Schwichtenberg (CAS)

B. Klatt (FZI)

Fertiggestellt am: 31.05.2013

Schlagworte: Frameworkentwicklung, Datenhaltung

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung
01	Alle	31.05.2013	Initiale Fassung

ToDos

Abschnitt	von	Beschreibung	Pritorität	Todo

Inhalt

Abbildungen

1 Einleitung

Im Rahmen des MOHITO Projektes soll ein Framework auf Basis einer Server- und einer Client Komponente mit ergänzenden Frameworks auf dem Stand der Technik entwickelt werden, welches den nicht-generierten Teil der Datenhaltungsschicht in plattformspezifischer Ausprägung vorhält, Offlinesynchronisation und – Datenhaltung ermöglicht sowie auf variable Konnektivität und mögliche Bandbreitenbeschränkung reagieren kann. Diese plattformspezifischen Ausprägungen des Frameworks können von verschiedenen Plattformen genutzt werden.

Grundlage für die hier beschriebene Framework Entwicklung sind zum einen die „CAS Open Server Components“ seitens der CAS, zum anderen der „MML“ seitens der B2M mit ihren entsprechenden Client-Anwendungen, auf welche im Folgenden detaillierter eingegangen werden soll.

2 CAS Open Server Components

Der CAS Open Server, der im Rahmen des MOHITO Projekts zum Einsatz kommt und im Zusammenspiel mit verschiedenen mobile Clients eine xRM Anwendung bildet, ist gemäß einer klassischen 3-Schichten Architektur entwickelt.

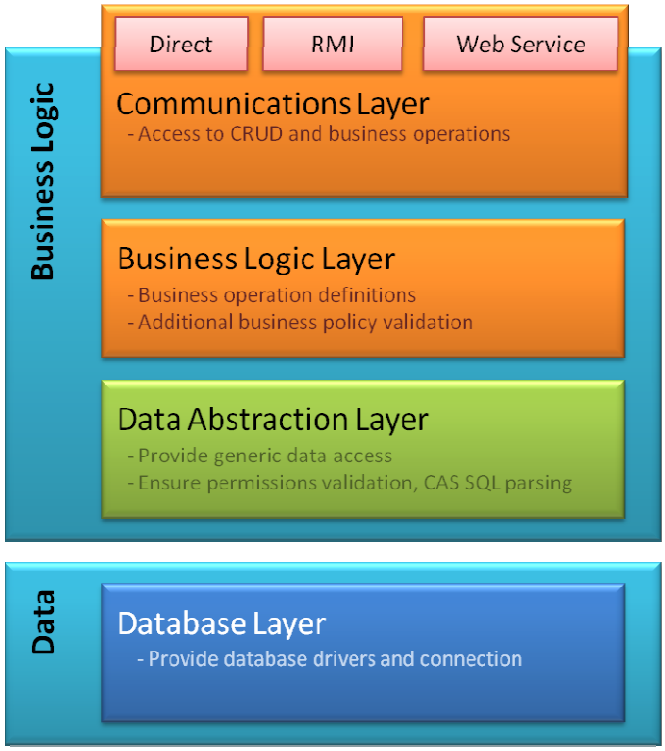

Abbildung 1 CAS Open Server Architektur

Die Schichten des Open Servers werden durch eine Vielzahl an Komponenten gebildet, die basierend auf den Technologien Java EE, Spring und OSGi modular entworfen sind. Die Komplexität dieser Technologien wird durch das EIMInterface verborgen, das Zugriffe auf die Business Logik und die Daten per SOAP, Remote Method Invocation (RMI) oder direkt ermöglicht. Zusätzlich stellt der Open Server ein REST Interface zur Verfügung, das wiederum auf das EIMInterface zugreift und einen großen Teil der Funktionalität des EIMInterfaces gemäß des REST Paradigmas bereitstellt. Die mobilen Clients, für die im Rahmen des MOHITO Projekts Datenhaltungsschichten geniert werden, kommunizieren ausschließlich via REST mit dem Server.
Die Kommunikationsschicht abstrahiert von der Business Logik Schicht, die wiederum die Datenzugriffsschicht abstrahiert. Die Kommunikationsschicht ermöglicht den Zugriff auf CRUD (Create Retrieve Update Delete) Operationen und Business Logik Operationen. Die Business Operationen sind in der Business Logik Schicht definiert und implementiert. Zusätzlich werden hier Business Policies validiert, z.B. wird geprüft, ob die vom Benutzer eingegebenen Daten valide sind. CRUD Operationen, also Operationen auf den Daten, die keine anwendungsspezifische Verarbeitung erfordern, werden quasi an der Business Logik Schicht vorbei, direkt auf der Datenzugriffsschicht ausgeführt. Die Datenzugriffsschicht übernimmt die Validierung der Zugriffsrechte (hier wird überprüft, ob der Benutzer überhaupt Zugriff auf die von ihm geforderten Daten hat). Die Datenzugriffsschicht abstrahiert von der eigentlichen Datenhaltung und stellt generische Mechanismen des Datenzugriffs zur Verfügung. Die Datenhaltungsschicht selbst stellt u. A. Datenbank Treiber und Datenbank Konnektoren zur Verfügung und setzt die SQL Anfragen ab, die in der Datenzugriffsschicht komponiert werden.

2.1 Mobile Apps und REST

Die REST-Schnittstelle unterstützt folgende HTTP-Methoden:

- GET zum Auslesen von Daten
- POST zum erstmaligen Erzeugen von Objekten
- PUT zum Ändern von Objekten
- DELETE zum Löschen von Objekten

Die PUT- und POST-Operationen akzeptieren in der Regel Daten im selben Format, wie die GET-Operationen es zurückgeben. Bei den PUT-Methoden (Update) muss ein Updatetimestamp mitgegeben werden.
Zur Übertragung werden die Daten als JSON serialisiert, der Content-Type ist also in der Regel application/json. Bei Dokumenten (PDF, Word) entspricht der Content-Type der Dateiendung (z.B. pdf, doc).

Ein beispielhafter POST Request zum Anlegen einer Adresse (Content-Type *application/json*), der an die Server URL https://login.cas-pia.com/pia/rest/v2.0/type/address geschickt wird sie folgendermaßen aus:


```
{
	"fields": {
		"COMPNAME": "CAS Software AG",
		"MAILFIELDSTR1": "info@cas.de",
		"STREET1": "CAS-Weg 1",
		"ZIP1": "76131",
		"TOWN1": "München",
		"ISORGANISATION": true,
		"FIRSTCONTACTDATE": "1999-03-25T00:00:00.000Z",
		"GWADDRESSFORMAT": "DE",
		"ADDRESSTERM": "Firma"
	}
}
```


Daraufhin würde der Server mit der URL antworten, unter der der neue Datensatz abgelegt wurde und unter der er wieder abgerufen werden kann:
https://login.cas-pia.com/pia/rest/v2.0/type/address/69E344D3DB7F3267ABBA259418162A90

Die REST API ist umfassend unter folgender URL beschrieben:
https://login.cas-pia.com/pia/rest/v2.0/help
Eine Übersicht aller Felder eines Objekttyps kann man über die GET-Operation GetSchema abgerufen werden. In der Auflistung der Felder sind deren Typen, die Anzeigenamen in der Oberfläche, mögliche Feldlängen sowie weitere Metainformationen angegeben. Hier werden auch vom Administrator definierte Felder mit ihren realen Feldnamen zurückgegeben, vgl. beispielsweise: https://login.cas-pia.com/pia/rest/v2.0/type/address/schema

2.2 CRUD Operationen

Die CRUD Operationen werden - wie bereits erwähnt – direkt in der Datenzugriffsschicht interpretiert und ausgeführt. Die CRUD Operationen werden initial von den sogenannten Data Access Objects (DAOs) behandelt. Die konkreten DAOs sind Subklassen von AbstractDataObjectDao. Eine wichtige Rolle der DAOs ist das Überprüfen der Datenkonsistenz, was von den sogenannten DAO-Plugins übernommen wird. Da das Datenmodell der CAS xRM Anwendung erweitert und kundenspezifisch angepasst werden kann, stellt der Open Server hierfür Funktionalität auf allen Anwendungsschichten zur Verfügung. So kön-

nen neue DAO Plugins implementiert und registriert werden, die für neue, spezifische Datentypen beispielsweise die Datenkonsistenz auf domänenspezifische Art und Weise prüfen. Die (De-) Registrierung wird von einem OSGi Service übernommen, bei dem über Spring Mechanismen neue DAO Plugins registriert werden können.
Der eigentliche JDBC Zugriff erfolgt weiter unten in der Datenhaltungsschicht über die DataProvider Objekte. Die Data Provider stellen ein Interface für die (prinzipiell austauschbare MySQL) Datenbank zur Verfügung und kapseln die verschiedenen SQL Dialekte, indem sie einen standardisierten Dialekt CAS SQL definierten. Der CAS SQL Parser interpretiert Anfragen, die in diesem CAS SQL Dialekt gestellt werden, und übersetzt sie in den entsprechenden konkreten SQL Dialekt unter Berücksichtigung der Zugriffsrechte und des Löschzustands einzelner Daten Objekte. Ferner analysiert der Parser spezielle CAS SQL Prädikate und sorgt für die entsprechenden Evaluationen.

2.3 Generisches Datenmodell

Wie bereits erwähnt, unterstützt der Open Server ein generisches Datenmodell, das kunden- oder branchenspezifisch angepasst werden kann. Dies ist gerade im Falle einer xRM Anwendung relevant, da ein Kunde aus dem Bereich Gesundheitswesen ganz andere Daten verwalten möchte, als beispielsweise ein Kunde der Versicherungsbranche, der Verträge und nicht Krankenakten verwaltet und damit auch ganz andere Attribute für diese Daten angeben und nachverfolgen möchte.
Die Datenobjekttypen (und deren Felder, wie zum Beispiel der Name einer Person) sind im Open Server also nicht hart kodiert, sondern werden kundenspezifisch angelegt.
Um die kundenspezifischen Daten Typen und entsprechenden Datenbanken anzulegen, wird im Open Server für jeden Daten-Objekt-Typ eine XML Datei angelegt, die die Felder und Metainformationen des Daten-Objekt-Typs beschreibt, z.B. ADDRESS.xml, APPOINTMENT.xml, etc. Basierend auf diesen XML Dateien werden auch Java Konstanten generiert. Die Server-Komponente DBAssistant liest die Datenbank-Schema-Definition aus der XML Datei und schreibt die entsprechenden Werte über Managementoperationen in die entsprechenden Datenbanktabellen (s. Abbildung 2).
Die kundenspezifischen Schema Meta Informationen werden zur Laufzeit vom Open Server interpretiert. Zusätzlich zu den kundenspezifischen Datenbankschemata existiert ein globales Schema, das als Einstiegspunkt dient. Im globalen Schema wird u. A. gespeichert, welche Kunden es gibt (zusammen mit deren Lizenzen) und es ist ein Mapping vom Kunden auf dessen Datenbank (inklusive des entsprechenden Datenbank Servers und der Authentifizierungsinformation) hinterlegt.
Da es keinen festen Satz an spezifischen Datenobjekttypen gibt, gibt es die bereits erwähnte generische Klasse DataObject. Jede konkrete Instanz hiervon korrespondiert mit einer Zeile

in der kundenspezifischen Datenbanktabelle, dessen Felder die Werte in den dazugehörigen Spalten sind.
Metadaten zu den Datenobjekten werden Java-seitig in DataObjectDescription Objekten repräsentiert.
Die folgende Grafik zeigt zum einen schematisch die Funktionalität des DBAssistant und außerdem wie sich die existierenden Tabellen des Kunden A ändern, wenn das neue Feld "CAR" hinzukommt.

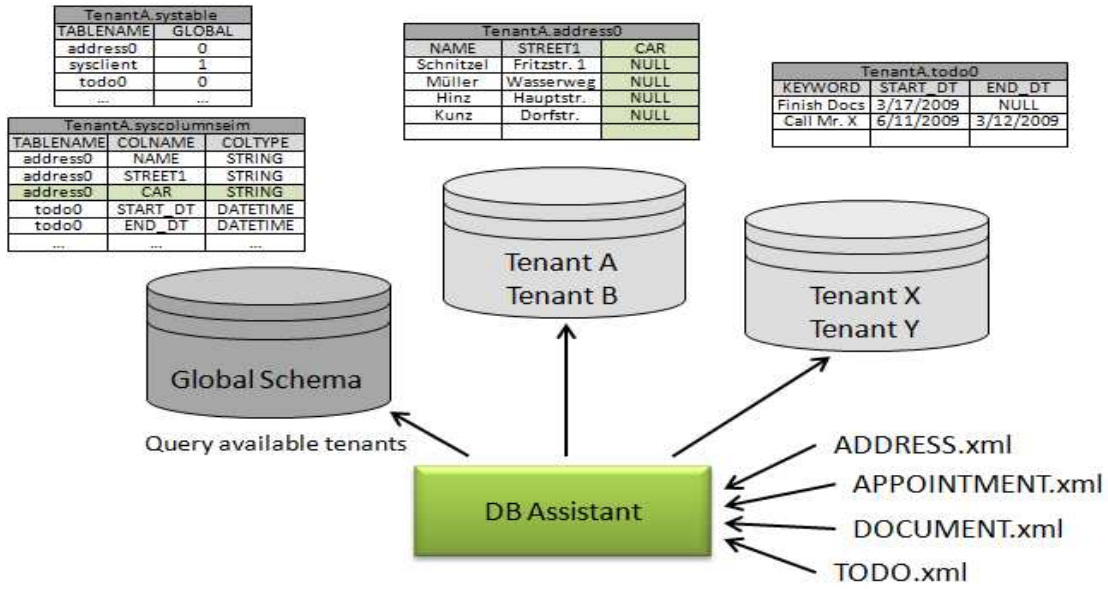

Abbildung 2 Funktionsweise des DBAssistant

2.4 Caching und Prefetching

Im Rahmen des MOHITO Projekts soll ein mobiler offline-fähiger xRM Client prototypisch entwickelt werden, dessen Datenhaltungsschicht für die mobilen Plattformen Android und iOS anhand einer Modellbeschreibung (in Form einer Domain Specific Language – DSL) generiert wird. Wesentliche Voraussetzung für die offline Funktionalität sind Caching- und Prefetching-Mechanismen. Caching meint hier das Speichern von bereits angeforderten Daten, Prefetching hingegen meint das Speichern von für den Benutzer relevanten Daten auf dem mobilen Endgerät, mit dem Ergebnis, dass diese gecachten und geprefetchten Daten auch offline verfügbar sind. Die gesamte Funktionalität für das Caching und Prefetching stellen die Clients zur Verfügung. Der Client weiß, welche Daten der Benutzer bereits abgerufen hat (Caching) und der Client setzt entsprechend der Prefetching Logik die Datenabfragen bei Server ab und speichert das Ergebnis (die relevanten Daten) lokal ab.

2.5 Synchronisation und Konfliktbehandlung

Die Benutzung von Clients im offline Modus kann prinzipiell zu Konflikten in den Datensätzen führen, da hier nicht zwangsläufig auf aktuellen Daten gearbeitet wird. Sobald der Client wieder online ist, werden die Daten im Rahmen einer Synchronisation abgeglichen. Die Synchronisation gleicht den Datenbestand des Clients mit dem des Servers ab. Der CAS Open Server setzt ein optimistisches Sperrverfahren ein, wodurch im Rahmen der Synchronisation Konflikte durch konkurrierende Änderungen auf Client und Server auftreten können. Für konkurrierende Änderungen müssen Konfliktlösungsstrategien existieren, die ggf. automatisch angewendet werden können.

		Server				
		Keine Aktion	Neu angelegt	Geändert	Gelöscht mit Undo-Option	Gelöscht ohne Undo-Option
Offline-Client	Keine Aktion	-	Auf Client erzeugen	Auf Client aktualisieren	Auf Client als gelöscht markieren	Auf Client löschen
	Neu angelegt	Auf Server erzeugen	-	-	-	-
	Geändert	Auf Server aktualisieren	-	(K1)	(K2)	(K3)
	Gelöscht mit Undo-Option	Auf Server als gelöscht markieren	-	(K4)	x	(K5)
	Gelöscht ohne Undo-Option	(nicht unterstützt)	(nicht unterstützt)	(nicht unterstützt)	(nicht unterstützt)	(nicht unterstützt)

*Tabelle 1 – Synchronisation eines Datensatzes zwischen Offline-Client und Server, Konfliktfälle K*n

Tabelle 1 stellt die bei der Synchronisation möglichen Konfliktfälle schematisch dar (1 – 5). Zur ihrer Auflösung können die folgende Konfliktlösungsstrategien angewendet werden.

Konfliktfall (K1)	Aktionen
Client gewinnt	Serverversion durch Clientversion ersetzen
Server gewinnt	Entfernen der Lösch-Markierung auf Server, Serverversion durch Client-version ersetzen
Gleichberechtigt	Merge auf Feldebene, Update Client- und Serverversion mit Merge-Ergebnis

Konfliktfall (K2)	Aktionen
Client gewinnt	Entfernen der Lösch-Markierung auf Server, falls gleichzeitig (1) vorliegt: Aktionen aus (1), falls nicht gleichzeitig (1): Serverversion durch Clientversion ersetzen
Server gewinnt	Löschmarkierung auf Client setzen, falls gleichzeitig (1) vorliegt: Aktionen aus (1), falls nicht gleichzeitig (1): Clientversion durch Serverversion ersetzen

Konfliktfall (K3)	Aktionen
Client gewinnt	Clientversion auf Server erzeugen
Server gewinnt	Clientversion löschen

Konfliktfall (K4)	Aktionen
Client gewinnt	Löschmarkierung auf Server setzen, falls gleichzeitig (1) vorliegt: Aktionen aus (1), falls nicht gleichzeitig (1): Serverversion durch Clientversion ersetzen
Server gewinnt	Löschmarkierung auf Client entfernen, falls gleichzeitig (1) vorliegt: Aktionen aus (1), falls nicht gleichzeitig (1): Clientversion durch Serverversion ersetzen,

Konfliktfall (K5)	Aktionen
Client gewinnt	Clientversion auf Server erzeugen, Löschmarkierung auf Server setzen
Server gewinnt	Clientversion löschen

3 B2M „HereIAm“–Client Komponenten

Die HereIAm Anwendung stellt im Zusammenspiel mit der MML Serverkomponente ein „Location Based Information“ System dar. Beide Komponenten, sowohl der Server als auch der Client, implementieren im Rahmen des MOHITO Projekts den Zugriff auf die entsprechenden Datenhaltungs-Stacks über das MOHITO Framework. Im Folgenden soll der Aufbau der Client- Anwendung und die daraus resultierende Ableitung der Schnittstelle für das MOHITO Basisframework in spezifischer Ausprägung für die Android Plattform erläutert werden.
Die Android basierte „HereIAm“ App stellt den Client des „Location Based Information“ Systems“ dar. Die Anwendung ist in ihrer Architektur in mehrere Schichten unterteilt. Zu Oberst befindet sich die Kommunikationsschicht. Sie ist für den Informationsaustausch mit dem Server verantwortlich. Die Kommunikationsschicht setzt auf das MOHITO Framework auf, auf dessen Aufbau ich später noch detaillierter eingehen möchte, da dies die zentrale Komponente dieses Dokumentes darstellt. Das MOHITO Framework ermöglicht einen einfachen, homogenen und plattformunabhängigen Zugriff auf die darunter liegende Datenhaltungsschichte. Diese Datenhaltungsschicht ist im Fall der Android Implementierung in eine „SQLite“ Datenbank, welche die eigentlichen Daten enthält und einen OR-Mapper, der den Zugriff auf diese erleichtert, unterteilt.

Abbildung 3 HereIAm Architektur

3.1 Kommunikationsschicht

Die Kommunikationsschicht ist für den Informationsaustausch mit dem Server verantwortlich. Die Kommunikation mit dem Server erfolgt über REST Anfragen. Die REST-Schnittstelle unterstützt hierbei alle gängigen HTTP-Methoden. Das sind:

- GET zum Auslesen von Daten
- POST zum erstmaligen Erzeugen von Objekten
- PUT zum Ändern von Objekten
- DELETE zum Löschen von Objekten

Bei Anfragen die komplexere Eingabeparameter benötigen wird JSON verwendet, um die benötigen Informationen zum Server zu übertragen. Die Antworten des Servers erfolgen entweder ebenfalls im JSON Format, oder bei extern angebundenen Diensten im SOAP Format, welche entsprechend geparst und verarbeitet werden.

3.2 Datenzugriffsschicht

Die Datenzugriffsschicht ist in die eigentliche Datenbank und einen OR-Mapper unterteilt. Als Datenbank kommt „SQLite“ zum Einsatz, was unter Android als De-facto-Standard angesehen werden kann. Um einen leichteren Zugriff auf die Datenbank zu ermöglichen und eine gewisse Konsistenz im Datenzugriff zu anderen Plattformen zu wahren, wird ein OR-Mapper verwendet, welcher auf der Datenbank aufsetzt und die relationalen Informationen der Datenbank auf Objektebene anhebt. Hier wird unter Android „OrmLite“ verwendet, da das Framework einen leichtgewichtigen OR-Mapper realisiert, welcher auch auf anderen Plattformen verfügbar ist. OrmLite übernimmt dann die eigentliche Kommunikation mit der „SQLite“ Datenbank über Android spezifische JDBC Treiberimplementierungen.

4 MOHITO Framework

Im Rahmen des MOHITO Projekts soll ein Multi-Plattform-Framework für Datenpersistenz und Datensynchronisation entstehen. Die Innovation von MOHITO gegenüber existierenden Ansätzen zur modellgetriebenen Generierung der Datenhaltungsschicht liegt insbesondere in der integrierten Unterstützung von server- und clientseitigen Mechanismen für Caching, Synchronisation und Datenpersistenz. Diese Funktionalitäten müssen von dem MOHITO Framework auf jeweils plattformspezifische Weise realisiert werden. Im Folgenden soll die Umsetzung des Frameworks auf der Android Plattform und die dabei entstandene generelle, plattformübergreifende Schnittstelle genauer beleuchtet werden. Die Schnittstellen des Frameworks sind dabei in die einzelnen Aufgabenbereiche Datenpersistenz und Datenzugriff, Caching und Synchronisation unterteilt.

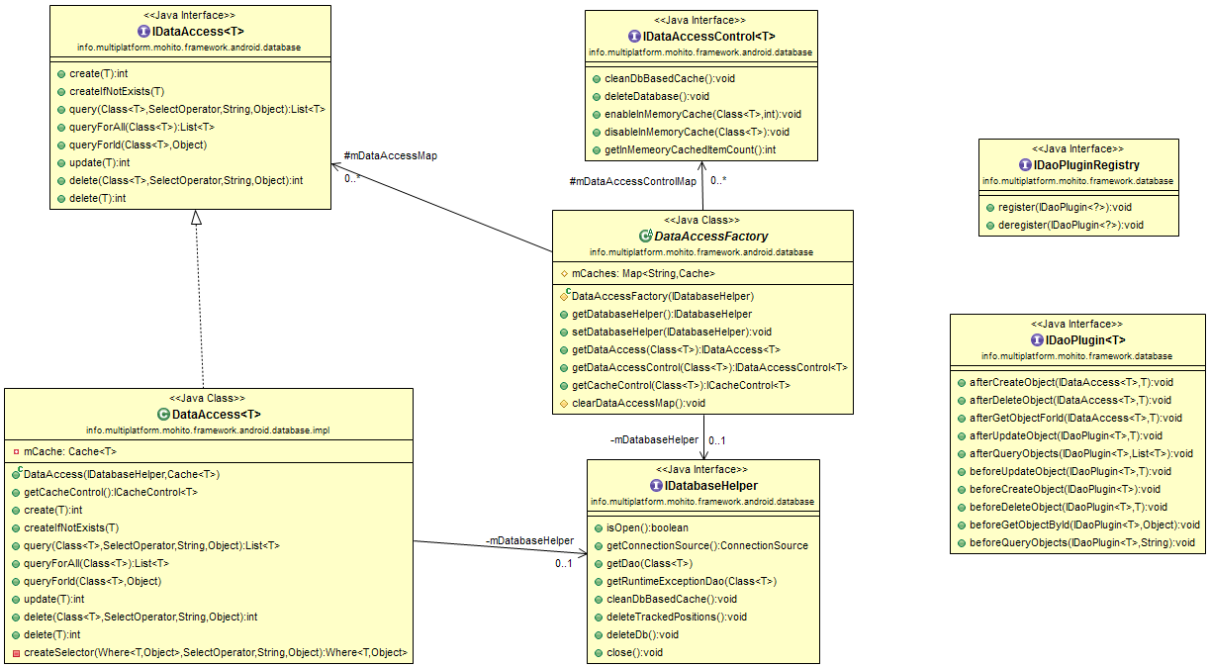

Abbildung 4 MOHITO – Schnittstellen zum Datenzugriff

4.1 Schnittstellen zum Datenzugriff

Das zuvor gezeigte UML Diagramm soll das generelle Zusammenspiel der einzelnen Komponenten veranschaulichen, welche einen homogenen und plattformübergreifenden Zugriff auf die zugrundeliegend Datenbanksysteme erlauben. Die Zentralen Schnittstellen hierbei sind IDataAccess, IDataAccessControl und der IDatabaseHelper.

4.1.1 IDataAccess Schnittstelle

Das IDataAccess Interface ermöglicht Datenoperationen auf persistierten Daten mit der Mächtigkeit von CRUD (create, read, update, delete).

Interface IDataAccess<T>

info.multiplatform.mohito.framework.android.database

---

public interface **IDataAccess<T>**

Method Summary		Page
int	**create**(T data) Use to store a new data entry in the underlying datastore.	12
T	**createIfNotExists**(T data) This is a convenience method for creating a data item, if the ID does not already exist.	12
int	**delete**(Class<T> clazz, SelectOperator operator, String fieldName, Object value) Deletes the objects, that match the given parameters.	14
int	**delete**(T data) Tries to delete the given object in the data store if it is existing	15
List<T>	**query**(Class<T> clazz, SelectOperator operator, String fieldName, Object value) Query for the items in the datastore, which match the given parameters.	13
List<T>	**queryForAll**(Class<T> clazz) This is a convenience method for querying for all of the items in the object table.	13
T	**queryForId**(Class<T> clazz, Object id) This is a convenience method for querying the item in the object table, having the given unique ID.	14
int	**update**(T data) Updates the given data item, if existing in the data store	14

Method Detail

create

int **create**(T data)
throws SQLException

Use to store a new data entry in the underlying datastore. It usually creates a new row in the database from a given object.

Parameters:

data - Data item to be created.

Returns:

The number of rows updated in the database. This should be 1.

Throws:

SQLException

---

createIfNotExists

T **createIfNotExists**(T data)
throws SQLException

This is a convenience method for creating a data item, if the ID does not already exist. The method extracts the ID from the data parameter, does a query for it and is returning the data if it exists. Otherwise a new data entry will be created with the given parameter.

Parameters:

data - Data item to be created if not existing.

Returns:

Either the data parameter if it was inserted, or the data element that already existed in the database.

Throws:

SQLException

---

query

List<T> **query**(Class<T> clazz,
`SelectOperator` operator,
String fieldName,
Object value)
throws SQLException,
`PersistencyException`

Query for the items in the datastore, which match the given parameters.

Parameters:

clazz - Class instance of the data item this method is searching for.
operator - The select operator used for the query.
fieldName - The fieldname in the underlying database, on which the query will be operating on.
value - Value which is used by the query in conjunction with the given select operator to determine the result set.

Returns:

The result set matching the query parameter.

Throws:

SQLException
`PersistencyException`

---

queryForAll

List<T> **queryForAll**(Class<T> clazz)
throws SQLException

This is a convenience method for querying for all of the items in the object table. NOTE: For medium sized or large tables, this may load a lot of objects into memory.

Parameters:

clazz - Class instance of the data item this method is searching for.

Returns:

The result set containing all items in the object table.

Throws:

SQLException

---

queryForId

T **queryForId**(Class<T> clazz,
Object id)
throws SQLException

This is a convenience method for querying the item in the object table, having the given unique ID.

Parameters:
clazz - Class instance of the data item this method is searching for.
id - Unique ID of the data item the method is searching for.
Returns:
The data item matching the given ID.
Throws:
SQLException

---

update

int **update**(T data)
throws SQLException

Updates the given data item, if existing in the data store

Parameters:
data - Data item to be updated if existing.
Returns:
The number of rows updated in the database.
Throws:
SQLException

---

delete

int **delete**(Class<T> clazz,
`SelectOperator` operator,
String fieldName,
Object value)
throws SQLException

Deletes the objects, that match the given parameters.

Parameters:

clazz - Class instance of the data item this method is trying to delete.

operator - The select operator used for the query.

fieldName - The fieldname in the underlying database, on which the query will be operating on.

value - Value which is used by the query in conjunction with the given select operator to determine the set of items to be deleted.

Returns:

The number of rows updated in the database.

Throws:

SQLException

---

delete

int **delete**(T data)
throws SQLException

Tries to delete the given object in the data store if it is existing

Parameters:

data - Data item to delete.

Returns:

The number of rows updated in the database. If the data item has been found and successfully deleted this should be 1.

Throws:

SQLException

4.1.2 IDataAccessControl Schnittstelle

Die IDataAccessControl Schnittstelle erlaubt Kontrolle über die Art und Weise, wie Daten Objekte im zugrundeliegenden Cache abgelegt werden.

Interface IDataAccessControl<T>

info.multiplatform.mohito.framework.android.database

---

public interface **IDataAccessControl<T>**

This class grants control over the way data items are stored in the underlying data store. The class enables you to clean or delete an underlying database or enable/disable in memory caching functionalities.

Method Summary		Page
void	**cleanDbBasedCache**() Use this method to clean the underlying data base by dropping and recreating all tables.	*16*
void	**deleteDatabase**() Use this method to completely delete the underlying data base.	*17*
void	**disableInMemoryCache**(Class<T> clazz) Disables the in-memory cache for the given class.	*17*
void	**enableInMemoryCache**(Class<T> clazz, int maxItems) Enables the in-memory cache for a certain number of items for the given class.	*17*
int	**getInMemeoryCachedItemCount**() Use this to get the number of items currently kept in the in-memory cache.	*18*

Method Detail

cleanDbBasedCache

void **cleanDbBasedCache**()
throws SQLException

Use this method to clean the underlying data base by dropping and recreating all tables.

Throws:

SQLException

---

deleteDatabase

void **deleteDatabase**()
throws SQLException

Use this method to completely delete the underlying data base.

Throws:

SQLException

---

enableInMemoryCache

void **enableInMemoryCache**(Class<T> clazz,
int maxItems)
throws SQLException

Enables the in-memory cache for a certain number of items for the given class.

Parameters:

clazz - Class instance of the data items, which should be kept in the in-memory cache.

maxItems - Maximum number of items being kept in the cache. Inserting an object into the cache once it is full will cause the least recently used object to be dropped.

Throws:

SQLException

---

disableInMemoryCache

void **disableInMemoryCache**(Class<T> clazz)
throws SQLException

Disables the in-memory cache for the given class.

Parameters:

clazz - Class instance of the data items, for which the caching should be disabled.

Throws:

SQLException

getInMemeoryCachedItemCount

int **getInMemeoryCachedItemCount**()

Use this to get the number of items currently kept in the in-memory cache.

Returns:

Number of items currently kept in the in-memory cache.

4.1.3 IDatabaseHelper Schnittstelle

Die IDatabaseHelper Schnittstelle ermöglicht direkten Zugriff auf die zugrunde liegende Datenbank.

Interface IDatabaseHelper

info.multiplatform.mohito.framework.android.database

public interface **IDatabaseHelper**

This class enables direct access to the underlying database.

Method Summary		Page
void	**cleanDbBasedCache**() Use to clean the cache, which is backed up by the underlying database.	20
void	**close**() Close any connections to the underlying database.	20
void	**deleteDb**() Delete the underlying database.	20
ConnectionSource	**getConnectionSource**() Retrieve a connection to the underlying database.	19
<D extends <any>,T> D	**getDao**(Class<T> clazz) Used to get direct access to a data access object (DAO), representing a data entry in the database.	19

<D extends <any>,T> D	**getRuntimeExceptionDao**(Class<T> clazz) Used to get direct access to a data access object (DAO), representing a data entry in the database.	*20*
boolean	**isOpen**() Check whether the underlying database is currently open, which means a connection to it is established.	*19*

Method Detail

isOpen

boolean **isOpen**()

Check whether the underlying database is currently open, which means a connection to it is established.

Returns:

True when there is an existing connection open to the database.

---

getConnectionSource

ConnectionSource **getConnectionSource**()

Retrieve a connection to the underlying database.

Returns:

connection to the underlying database.

---

getDao

<D extends <any>,T> D **getDao**(Class<T> clazz)
throws SQLException

Used to get direct access to a data access object (DAO), representing a data entry in the database.

Parameters:

clazz - Class instance of the stored data item.

Returns:

Returns a data access object (DAO) for the given class instance.

Throws:

SQLException

---

getRuntimeExceptionDao

<D extends <any>,T> D **getRuntimeExceptionDao**(Class<T> clazz)

Used to get direct access to a data access object (DAO), representing a data entry in the database. This method does the same as getDao but is not throwing any runtime exceptions on failures.

Parameters:

clazz - Class instance of the stored data item.

Returns:

Returns a data access object (DAO) for the given class instance.

---

cleanDbBasedCache

void **cleanDbBasedCache**()

Use to clean the cache, which is backed up by the underlying database.

---

deleteDb

void **deleteDb**()

Delete the underlying database.

---

close

void **close**()

Close any connections to the underlying database.

4.2 Die Cache Implementierung

Im Bereich der Datenbanksysteme haben sich semantische Caching Strategien als sinnvoll erwiesen. Beim semantischen Caching werden zu den gepufferten Daten zusätzliche Informationen gespeichert, die diese Daten beschreiben, sogenannte Metadaten. Bei Datenbankanfragen können so zu den erhaltenen Ergebnisdaten die zugehörigen Anfragen gespeichert werden, welche eine semantische Beschreibung der Antwortdaten darstellen. Bei der Anfragebearbeitung kann dann mithilfe der semantischen Informationen entschieden werden, ob eine Anfrage ganz, oder zumindest teilweise, aus dem Cache beantwortet werden kann. Wird eine identische Anfrage im Cache gefunden, kann diese direkt zurückgeliefert werden. Ist nach semantischer Analyse der im Cache vorhandenen logischen Informationen eine Teilmenge der gestellten Anfrage vorhanden, muss nur noch eine komplementäre Anfrage an die Datenbank gestellt werden, um die Antwort zu vervollständigen. In beiden Fällen wird Bandbreite, die sonst zum Übertragen der vollständigen Antwort benötigt worden wäre, eingespart.

Eine besondere Variante des Caching, die vor allem für mobile Geräte entwickelt wurde, ist das sogenannte „Data Hoarding“ (auch als Prefetching bezeichnet). Es bezeichnet das lokale Vorhalten von Daten für einen späteren Gebrauch. So werden Daten, die während späterer Offline-Phasen benötigt werden, in Situationen mit guter Konnektivität oder an dedizierten Orten, wie Terminals, die über schnelle WLAN Verbindungen verfügen, auf das Gerät geladen. Wichtig ist hierbei die passende Auswahl geeigneter Daten. Normalerweise ist es aufgrund der begrenzten Cachegröße nicht möglich alle Daten vorzuhalten. Daher ist aufgrund des Nutzerverhalten und der Nutzerpräferenzen, durch Heuristiken, oder über das Wissen einer zu erwartenden Offline-Situation, eine passende Datenauswahl zu treffen. Diese Daten ermöglichen dann in Offline-Situation ein autonomes Arbeiten.

Im Rahmen des Mohito Projektes haben wir einen beide Caching Varianten umgesetzt. Im Folgenden sollen die Schnittstellen vorgestellt werden, welche die entsprechenden Funktionalitäten zur Verfügung stellen.

4.2.1 ICacheControl Schnittstelle

Die ICacheControl Schnittstelle realisiert den Zugriff auf den Cache.

Interface ICacheControl<T>

info.multiplatform.mohito.framework.android.cache

---

public interface **ICacheControl<T>**

Method Summary		Page
Result	**bulkSaveData**(List<T> dataItems, ISemanticDescriptor semanticDescriptor) This is a convenience function to store a set of data items in the cache at once.	24
Result	**bulkSavePrefetchedData**(List<T> dataItems, ISemanticDescriptor semanticDescriptor) This is a convenience function to store a set of data items in the cache at once and mark them as pre-fetched.	24
void	**clearCache**() Completely cleans the cache by deleting all items in it.	13
ICacheDescriptor	**getCacheDescriptor**() Used to retrieve the cache descriptor, holding all meta information related to this cache instance.	14
void	**removeData**(ISemanticDescriptor semanticDescriptor) Removes all data items belonging to one semantic cache region.	12
List<T>	**retrieveData**(ISemanticDescriptor semanticDescriptor) Retrieves all data items belonging to one semantic cache region.	12
Status	**saveData**(T data, ISemanticDescriptor semanticDescriptor) Saves a data entry of a generic type in the cache.	23
Status	**savePrefetchedData**(T data, ISemanticDescriptor semanticDescriptor) Saves a data entry of a generic type as a pre-fetch data item in the cache.	23

void	**setCacheStrategy**(ICacheStrategy cacheStrategy) Used to set a cache strategy the cache should operate on to select items which should be deleted when the cache is growing over it's cache limit.	*13*
void	**setConflictResolver**(IConflictResolver<T> conflictResolver) Use this method to set the conflict resolver instance, which should be used to solve occurring merge conflicts.	*14*

Method Detail

saveData

Status **saveData**(T data, ISemanticDescriptor semanticDescriptor)
throws PersistencyException, InvalidParameterException

Saves a data entry of a generic type in the cache.

Parameters:

data - Data entry to be stored in the cache.

semanticDescriptor - As the cache is realized as a semantic cache, the semantic descriptor is used to store and retrieve data from the cache.

Returns:

The status of the save process. In the case, that the data item did not already exist in the cache, the new data is just added. In all other scenarios the new data item has to be merged with an already existing one. The merge process itself is handled by the IConflictResolver implementation, which can be set via a call to setConflictResolver(IConflictResolver conflictResolver).

Throws:

PersistencyException, InvalidParameterException

---

savePrefetchedData

Status **savePrefetchedData**(T data, ISemanticDescriptor semanticDescriptor)
throws PersistencyException, InvalidParameterException

Saves a data entry of a generic type as a pre-fetch data item in the cache. Data marked as pre-fetched will NOT get victim of any cache replacement strategy and therefore will not be automatically deleted from the cache.

Parameters:

data - Data entry to be stored and marked as pre-fetched in the cache.

semanticDescriptor - As the cache is realized as a semantic cache, the semantic descriptor is used to store and retrieve data from the cache.

Returns:

The status of the save process. In the case, that the data item did not already exist in the cache, the new data is just added. In all other scenarios the new data item has to be merged with an already existing one. The merge process itself is handled by the IConflictResolver implementation, which can be set via a call to `setConflictResolver(IConflictResolver conflictResolver)`.

Throws:

PersistencyException, InvalidParameterException

---

bulkSaveData

`Result` **bulkSaveData**(List<T> dataItems, `ISemanticDescriptor` semanticDescriptor) throws PersistencyException, InvalidParameterException

This is a convenience function to store a set of data items in the cache at once.

Parameters:

dataItems - List of data items to be stored in the cache.

semanticDescriptor - As the cache is realized as a semantic cache, the semantic descriptor is used to store and retrieve data or complete data sets from the cache.

Returns:

The result of the save process for all given items. The result object consists of three lists. The SuccessfullMerges lists contains all items, which could be successfully merged. The ConflictingMerges lists contains all items, which could not be successfully merged and probably need user interaction to solve the conflict. The Errors lists contains all items, for which an error occurred during the merge process. An error could e.g. be, that an item could not be accessed in the data base because of missing access rights. The merge process itself is handled by the IConflictResolver implementation, which can be set via a call to `setConflictResolver(IConflictResolver conflictResolver)`.

Throws:

PersistencyException, InvalidParameterException

---

bulkSavePrefetchedData

`Result` **bulkSavePrefetchedData**(List<T> dataItems, `ISemanticDescriptor` semanticDescriptor) throws PersistencyException, InvalidParameterException

This is a convenience function to store a set of data items in the cache at once and mark them as pre-fetched.

Parameters:

dataItems - List of data items to be stored in the cache.

semanticDescriptor - As the cache is realized as a semantic cache, the semantic descriptor is used to store and retrieve data or complete data sets from the cache.

Returns:

The result of the save process for all given items. The result object consists of three lists. The SuccessfullMerges lists contains all items, which could be successfully merged. The ConflictingMerges lists contains all items, which could not be successfully merged and probably need user interaction to solve the conflict. The Errors lists contains all items, for which an error occurred during the merge process. An error could e.g. be, that an item could not be accessed in the data base because of missing access rights. The merge process itself is handled by the IConflictResolver implementation, which can be set via a call to `setConflictResolver(IConflictResolver conflictResolver)`.

Throws:

PersistencyException, InvalidParameterException

---

retrieveData

List<T> **retrieveData**(`ISemanticDescriptor` semanticDescriptor)
throws info.multiplatform.mohito.framework.android.database.PersistencyException, InvalidParameterException

Retrieves all data items belonging to one semantic cache region.

Parameters:

semanticDescriptor - Describing the semantic cache region the data should be retrieved from.

Returns:

Returns a set of data items belonging to the semantic cache region given by the semanticDescriptor parameter.

Throws:

PersistencyException, InvalidParameterException

---

removeData

void **removeData**(`ISemanticDescriptor` semanticDescriptor)
throws info.multiplatform.mohito.framework.android.database.PersistencyException,
InvalidParameterException

Removes all data items belonging to one semantic cache region.

Parameters:

semanticDescriptor - Describing the semantic cache region, that should be removed from the cache. This means all data items belonging to that region will be removed too.

Throws:

PersistencyException, InvalidParameterException

---

clearCache

void **clearCache**()

Completely cleans the cache by deleting all items in it.

---

setCacheStrategy

void **setCacheStrategy**(`ICacheStrategy` cacheStrategy)

Used to set a cache strategy the cache should operate on to select items which should be deleted when the cache is growing over it's cache limit. The Mohito framework provides implementations of the common strategies: FIFO, LFU, LRU, and MRU.

---

setConflictResolver

void **setConflictResolver**(`IConflictResolver`<T> conflictResolver)

Use this method to set the conflict resolver instance, which should be used to solve occurring merge conflicts. If no conflict resolver is set a default resolver implementation will be used for resolving potential conflicts during a data synchronization process.

---

getCacheDescriptor

`ICacheDescriptor` **getCacheDescriptor**()

Used to retrieve the cache descriptor, holding all meta information related to this cache instance.

Returns:

Returns the cache descriptor, holding all meta information related to this cache instance.

4.2.2 ICacheDescriptor Schnittstelle

Die ICacheDescriptor Schnittstelle hält alle relevanten Meta-Daten über eine Cache Instanz vor.

Interface ICacheDescriptor

info.multiplatform.mohito.framework.android.cache

---

public interface **ICacheDescriptor**

Holds meta data belonging to one cache instance.

Method Summary		Page
String	**getExternalSemanticFieldDescriptorColumn**()	28
Class<?>	**getExternalSemanticFieldItemType**()	16
String	**getForeignKeyColumn**()	15
String	**getIdColumnOfCachedItem**()	14

Method Detail

getIdColumnOfCachedItem

String **getIdColumnOfCachedItem**()

Returns:

Returns the column, which is used to uniquely identify one cache item.

---

getForeignKeyColumn

String **getForeignKeyColumn**()

Returns:

Returns the foreign key column.

---

getExternalSemanticFieldDescriptorColumn

String **getExternalSemanticFieldDescriptorColumn**()

Returns:

Returns the column, which is used as the semantic field descriptor.

---

getExternalSemanticFieldItemType

Class<?> **getExternalSemanticFieldItemType**()

Returns:

Returns a class instance of the external semantic field.

4.2.3 ICacheStrategy Schnittstelle

Das ICacheStrategy Interface repräsentiert nötige Funktionalitäten zur Implementierung einer Cache Verdrängungsstrategie.

Interface ICacheStrategy

info.multiplatform.mohito.framework.android.cache

---

public interface **ICacheStrategy**

Represents the implementation of a cache replacement strategy.

Method Summary		Page
<U> U	`getSemanticFieldToDelete`(Class<U> clazz) Selects the semantic filed that should be victim of the according cache replacement strategy.	17

Method Detail

getSemanticFieldToDelete

<U> U **getSemanticFieldToDelete**(Class<U> clazz)
throws UnsupportedOperationException

Selects the semantic filed that should be victim of the according cache replacement strategy.

Parameters:

clazz - Class instance of the data item the cache operates on.

Returns:

Returns the semantic field selected for deletion according the implemented cache replacement strategy.

Throws:

UnsupportedOperationException

4.2.4 Datensynchronisation und Konfliktlösungsstrategien

Im Bereich der Datensynchronisation gibt es etliche Verfahren um das Arbeiten mehrerer Systeme auf gemeinsamen Daten zu ermöglichen. Eine Möglichkeit ist auf die Replikation der Daten generell zu verzichten und nur eine zentrale Datenquelle zu haben, auf welcher alle Anwendungen über entsprechende Kommunikationsmechanismen arbeiten. Grundvoraussetzung eines solchen Systems ist allerdings eine hochverfügbare und performante Netzwerkinfrastruktur und die Bedingung, dass alle beteiligten Klienten jederzeit Teil dieses Netzwerks sind. Diese Vorrausetzung ist allerdings im Rahmen des MOHITO Projekts explizit nicht gegebene. Hier soll es im Gegenteil möglich sein, dass Anwendungen, welche sich auf mobilen Klienten befinden, im Fall einer eingeschränkten oder nicht vorhandenen Netzwerkverbindung weiterhin fehlerfrei auf den gemeinsamen Daten arbeiten können. In solch einem Umfeld müssen die die gemeinsamen Daten repliziert werden, um sie auch auf Geräten, welche temporär nicht mit dem Netzwerk verbunden sind, verfügbar zu machen. Mit einer Datenreplikation geht aber auch immer die Notwendigkeit einher die Daten synchron zu halten bzw. sie in regelmäßigen Abständen zu synchronisieren.

Bei solch einer Datensynchronisation kann zu Konflikten zwischen einzelnen Datensätzen kommen, die entsprechend aufgelöst werden müssen, um die Konsistenz einer Replikationsumgebung zu gewährleisten. Hierzu muss das MOHITO Framework entsprechende Konfliktlösungsstrategien vorhalten. Diese sind über die IConflictResolver Schnittstelle realisiert. Entsprechende Implementierungen dieser Schnittstellen können unterschiedliche Strategien umsetzen. So wäre es möglich zu versuchen Konflikte durch Automatismen aufzulösen. In diesem Bereich kommen häufig regelbasierten Prioritätsverfahren zum Einsatz. Bei solchen Verfahren wird ein Konflikt über definierte statische Regeln aufgelöst. So kann eine Regel beispielweise festlegen, dass im Konfliktfall immer der neuere Datensatz gewinnt. Es kann aber auch eine Prioritätsfolge für bestimmte Replikationsknoten festgelegt werden, die entscheidet dass Datensätze von einer bestimmten Quelle anderen gegenüber bevorzugt werden (Trust your Friends). Stehen mehr als zwei Quellen zur Auswahl kann der Konflikt auch über ein Mehrheitsverfahren aufgelöst werden, wobei derjenige Datensatz gewinnt, der häufiger vorkommt. Generell ist das automatisierte Auflösen von Konflikten fehlerbehaftet, da nicht immer durch statische Regeln im Vorfeld entschieden werden kann, welcher Datensatz in einer gewissen Situation der richtige ist. Daher existieren auch Ansätze bei denen der Benutzer über geeignete Benutzerschnittstellen bei der Auflösung eines Konflikts involviert wird, da er oft am besten entscheiden kann, wie ein Konflikt zu lösen ist.

Im Folgenden wird nur die IConflictResolver Schnittstelle vorgestellt. Entspreche Implementierungen, welche unterschiedliche Konfliktlösungsstrategien realisieren sind dann Bestandteil von Deliverable L 6.2.1 (Basisframework).

4.2.4.1 IConflictResolver Schnittstelle

Die IConflictResolver Schnittstelle definiert die nötigen Funktionalitäten, um Konflikte aufzulösen, die während einer Datensynchronisation aufgetreten sind.

Interface IConflictResolver<T>

info.multiplatform.mohito.framework.android.cache

---

public interface **IConflictResolver<T>**

Implementations of this interface are destined to solve merge conflicts, which can occur when cached data is synchronized with the server.

Method Summary		Page
Status	**handleConflict**(T existingDataItem, T dataItemToMerge) This method is called to handle merge conflicts.	31

Method Detail

handleConflict

Status **handleConflict**(T existingDataItem,
T dataItemToMerge)

This method is called to handle merge conflicts.

Parameters:

existingDataItem - This is the existing data item, which should be merged against the one to add.

dataItemToMerge - This is the new data item, which should be merged against the already existing one.

Returns:

The returned status represents the result of the conflict handling.

5 Zusammenfassung

Durch die vorangegangen Analyse von sowohl Server als auch Client-Komponenten, war es möglich ein homogenes Framework zur Datenhaltung, zum Caching und zur Offlinesynchronisation von Daten unter Berücksichtigung variabler Konnektivität und entsprechenden Bandbreitenbeschränkungen zu entwickeln. Clientseitig stand hierbei der korrekte Umgang mit ständig wechselnder Konnektivität oder vollständigen Offline Szenarien im Vordergrund, wohingegen Serverseitig eher eine performante Datenhaltung mit geeigneten Zugriffsmechanismen in den Fokus rückten.

Die, in den vorangegangenen Kapiteln detailliert dargestellten Schnittstellen, liegen in der momentanen Form in einer plattformspezifischen Ausprägung in Java vor. Es ist allerdings recht einfach möglich die entsprechenden Interfaces und deren Implementierungen auf andere Plattformen und andere Programmiersprachen zu portieren, um auch dort Funktionalitäten des MOHITO Frameworks zur Verfügung zu stellen.